Pejvan Beigui

LE GUIDE DE SURVIE

Objective-C 2.0

LE LANGAGE DE PROGRAMMATION IPHONE ET COCOA SUR MAC OS X



PEARSON

Objective-C 2.0

Pejvan Beigui

Publié par Pearson Education France

47 bis, rue des Vinaigriers

75010 PARIS

Tél. : 01 72 74 90 00

www.pearson.fr

Collaboration éditoriale : Digit Books

Réalisation PAO : Léa B.

ISBN : 978-2-7440-4126-6

Table des matières

Annexes

À propos de l'auteur

Doté d'une double compétence en informatique et en produits et marchés financiers, Pejvan est développeur et chef de projet dans une banque d'investissement américaine à Londres où il participe au développement d'outils de trading et de gestion des risques essentiellement en C# et plus récemment Python.

Avant de partir pour Londres, il a travaillé pour Apple en tant qu'employé, puis consultant dans l'équipe des relations développeurs où il donnait des conférences et formait les développeurs et partenaires aux technologies Apple (Carbon, puis Cocoa). Il a également fondé l'association, puis la société *Project:Omega* (**www.projectomega.org**, **www.projectomega.com**) avec son ami Thierry.

Introduction

Objective-C est un langage popularisé par Apple, société qui a connu sa renaissance depuis le retour en 1997 de Steve Jobs aux commandes. Il avait créé cette entreprise une vingtaine d'année plus tôt avant de se faire licencier et de partir fonder NeXT. Objective-C et NeXTStep sont à la base du succès de Mac OS X et de l'iPhone.

Bien que NeXT, puis Apple, soient les entreprises ayant popularisé Objective-C, le langage a été créé au début des années 80 par Stepstone, société de Brad Cox[1] et Tom Love. En 1988, NeXT a acquis une licence auprès de Stepstone afin de créer son propre compilateur, puis ses propres bibliothèques pour créer l'environnement NeXTstep[2] (d'où le fameux préfixe NS des classes des *frameworks Foundation* et *AppKit :* NS est l'acronyme de NeXTStep).

1. Merci à Jimmy Stewart de m'avoir appris ce fait.

2. Voir l'entrée Objective-C de Wikipedia.

Objective-C a su rester simple et changer relativement peu au fil des années tout en continuant de répondre au besoin des développeurs, comme le prouve le phénoménal succès d'Apple. En octobre 2007, Apple a publié une nouvelle version du langage : Objective-C 2.0, version qui a donné un nouveau souffle à Objective-C en le modernisant.

Objectif de ce livre

Il n'existe pas aujourd'hui suffisamment de ressources en français sur Objective-C, le principal langage de programmation des plateformes Apple : Mac et iPhone. C'est pourquoi, j'ai été très heureux et honoré de m'être vu proposer d'écrire ce livre, qui je l'espère comblera les développeurs qui souhaitent disposer d'un guide de survie sur ce langage.

En écrivant ce livre, j'ai essayé de satisfaire plusieurs besoins plus ou moins opposés : le format des guides de survie impose une approche très pragmatique du langage, des extraits et exemples de code à quasiment chaque section, et c'est une très bonne chose.

Toutefois, Objective-C étant un langage relativement distinct des autres, j'ai essayé d'introduire également les aspects un peu plus théoriques du langage, nécessaires à la bonne compréhension des paradigmes et à l'écriture d'un code sans erreurs et de bonne qualité.

J'ai également cherché à m'adresser non seulement au développeur débutant, qui n'a jamais programmé en Objective-C, mais également à aider le développeur Objective-C expérimenté souhaitant franchir le pas vers Objective-C 2.0.

De plus, de nombreux développeurs souhaitent migrer, ou complémenter leur offre logicielle, et améliorer leurs compétences : disposer d'une version Mac d'un logiciel Java ou Windows est désormais largement à l'ordre du jour, et proposer une version iPhone est devenu quasiment obligatoire.

C'est pourquoi j'ai essayé d'aider les développeurs expérimentés à faire une transition sans douleur en comparant Objective-C à Java et C# autant que j'ai pu, et, dans une moindre mesure, à C++ et Python.

Enfin, Objective-C repose beaucoup sur les bibliothèques Cocoa (Mac OS X), Cocoa Touch (iPhone) et se limiter à 100 % au langage aurait été non seulement difficile, mais aurait sans doute privé le lecteur de nombreuses connaissances nécessaires à la programmation de logiciels sur ces plateformes.

Les objectifs de cet ouvrage ont donc été très ambitieux et il a souvent fallu faire des choix difficiles. J'espère toutefois que vous y trouverez ce que vous cherchiez et qu'il satisfera vos besoins.

Organisation de ce livre

Ce livre n'est pas une référence sur Objective-C, mais il introduit toutefois tous les concepts nécessaires à l'écriture de code Objective-C de qualité. Il n'est pas destiné à être lu de manière linéaire, même si cela reste possible, voire recommandé selon votre niveau de compétences en Objective-C.

Bien qu'il n'y ait pas vraiment de pré-requis à la lecture de ce livre, Objective-C est un langage basé sur C, et par conséquent, une connaissance de ce dernier est plus ou moins nécessaire. De plus, même si vous n'avez jamais programmé en C, il n'est pas utile d'en apprendre les rudiments.

Le premier chapitre introduit Objective-C, les concepts de base du langage dans sa mouture 1.0 : les éléments, les mots-clés et les concepts.

Le deuxième chapitre s'attaque à la gestion manuelle de la mémoire, qui est toujours un sujet difficile en Objective-C et nécessaire si vous programmez sur iPhone. C'est ici que le développeur doit faire le plus attention, car une mauvaise gestion de la mémoire ne pardonne pas en Objective-C : c'est soit le plantage immédiat, soit la fuite de mémoire. Et si Objective-C est un langage relativement simple d'accès, la gestion de la mémoire reste complexe.

Le troisième chapitre est dédié à Objective-C 2.0 et peut être lu de manière indépendante pour les développeurs déjà familiers avec Objective-C 1.0. Il aborde les changements apparus entre la version 1 et 2, ainsi que les nouveautés de la version 2.0.

Le quatrième chapitre présente les notifications et les événements, deux concepts distincts en Objective-C et nécessaires pour faire communiquer vos classes entre elles et avec le matériel (le Mac ou l'iPhone).

Le cinquième chapitre est consacré à la qualité de code : gestion des exceptions et des erreurs, assertions, historique d'exécution, mais également les tests unitaires, que je considère comme extrêmement importants pour tout projet de développement adapté au XXIe siècle.

Enfin, deux annexes terminent ce livre. L'une est consacrée à Objective-C++, un langage né de la pseudo-fusion de C++ et Objective-C, l'autre aux ressources utiles, où je partage les liens et les outils que j'utilise le plus souvent.

Pour conclure, ce livre se voulant également un aide-mémoire, il serait incomplet sans de nombreux exemples. Nous ne vous recommanderons jamais assez de lire le code des autres pour voir des styles de programmation que nous n'abordons pas dans cet ouvrage. C'est un objectif que je me suis également fixé dans ce livre : vous montrer sans prétention comment je programme, dans l'idée que vous

améliorerez votre style en critiquant le mien. Aussi, les exemples de ce livre sont à l'image de mes programmes, certains étant même indirectement issus de ceux-ci.

Remerciements

Je souhaite remercier les Éditions Pearson et Patricia Moncorgé pour m'avoir permis de vivre l'aventure qu'a été la rédaction de cet ouvrage, ainsi que Dominique Buraud pour sa patience et son aide au cours des nombreux mois, ayant permis d'aboutir au texte que vous êtes en train de lire.

Et bien sûr, Noras, Anahita, Ardavan, Kouchan, Ferechteh, toute ma famille et mes amis qui m'ont soutenu (voire supporté) et encouragé pendant la rédaction et à qui je n'ai pas pu consacrer le temps que j'aurais dû au cours des derniers mois ; je les remercie de leur patience.

1

Les bases d'Objective-C

Objective-C est un langage de programmation orienté objet basé sur le langage C, tout comme C++. Cette spécificité intéressante rend le langage compatible avec C et dans une certaine mesure avec C++. Objective-C est également un langage au typage dynamique, contrairement à C++. Il dispose en conséquence d'un environnement d'exécution dédié, similaire par exemple à Java ou C#, mais toutefois plus réduit en termes de compétences. De plus, contrairement à Java et C#, Objective-C est compilé directement en langage machine et non en langage intermédiaire.

Dans ce chapitre, nous abordons ce qui est indispensable : les éléments, les mots-clés et les concepts du langage. Chaque section sera étayée de plusieurs exemples.

Définition de id

```
id number = [NSNumber numberWithInt:5];
```

Le code ci-dessus montre qu'un objet est déclaré sans typage avec le mot-clé `id`.

Objective-C est un langage dynamique, mais il autorise une vérification des références lors de la compilation. Ces notions seront abordées plus loin, mais il est toutefois important de remarquer qu'Objective-C se rapproche davantage de Python, Ruby et des autres langages dynamiques que de Java ou C# dans ce domaine.

Objective-C permet d'introduire un objet sans pour autant préciser son type grâce au mot-clé `id` qui peut être lu "identifiant d'objet" et ne force aucun typage. `id` est donc un pointeur vers un objet. Plus précisément, `id` est défini comme un pointeur vers la structure de données de l'objet.

Voici un exemple pour bien se rendre compte de ce fait – vous pouvez ignorer pour l'instant le reste de la syntaxe qui est toutefois simple à comprendre – les deux méthodes, `exemple1` et `exemple2` sont toutes les deux de type `id`, mais renvoient deux objets de type différents :

Exemple 1

```
exemple1() {
   //le type renvoyé n'est pas spécifié
   //il n'y a pas d'avertissement du compilateur
   //pourtant la méthode renvoie un 'id' de
   //type NSNumber
   id number = [NSNumber numberWithInt:5];
   return number;
}
```

Exemple 2

```
exemple2(){
    id date = [NSDate date];
    return date;
}
```

Autre point important à noter : `id` est le type par défaut de la valeur retournée par n'importe quelle méthode Objective-C. Ceci est logique dans la construction d'Objective-C, mais diffère du type par défaut retourné par les fonctions en C où le type retourné par défaut est `int`.

```
exemple1(){
  warning: return type defaults to 'int'
    return 1;
}
```

Figure 1.1 : Avertissement du compilateur, qui signale que le type par défaut de la fonction sera `int`.

Pour aller plus loin

Objective-C étant construit sur les fondations de C, il est possible de creuser un peu les éléments qui forment le cœur du langage. Par exemple, `id` est tout simplement défini comme un pointeur vers une structure `struct`, comme précisé dans le fichier d'en-tête `/Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h` :

```
typedef struct objc_object {
    Class isa;
} *id;
```

Objet-classe

```
Class uneClasse = [unObjet class];
```

À la différence de la plupart des langages, une classe Objective-C est également un objet, mais un objet de type particulier. C'est ce que l'on appelle un *objet-classe*. C'est grâce à cette particularité qu'il est possible d'utiliser un nom de classe comme type lorsqu'un objet est déclaré.

De plus, Objective-C définit également un type particulier pour les objets-classes : `Class`, qui n'est autre qu'un pointeur vers une structure opaque de type `objc_class`.Vous trouverez ces définitions dans

le fichier d'en-tête /Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h :

```
typedef struct objc_class *Class;
```

De plus, nous pouvons clarifier davantage la définition de id donnée à la section précédente :

```
typedef struct objc_object {
    Class isa;
} *id;
```

id est un pointeur vers une structure opaque de type objc_object (qui définit la structure des objets Objective-C). Chaque objet possède donc ce que l'on appelle un pointeur isa qui pointe vers un objet de type Class. Comme vous l'avez sans doute deviné maintenant, chaque objet pointe donc *via* son pointeur isa vers son objet-classe.

Autre point intéressant à noter : id peut donc également représenter un objet-classe.

Déclarer un objet

```
id declarationTypageDynamique;
ClasseGuideDeSurvie *declarationTypageStatique;
```

Il existe deux sortes de déclarations en Objective-C :

- Le typage statique. On précise le type de l'objet, c'est-à-dire la classe à laquelle appartient l'objet,

au moment de la déclaration. Le compilateur s'assure que ce type est respecté tout au long du code.

- **Le typage dynamique.** On déclare l'objet sans préciser son type, grâce au pointeur `id`. Le compilateur ne possède alors aucune connaissance sur le type de la référence, qui peut désormais représenter une instance de n'importe quelle classe.

Dans la grande majorité des cas, le type de l'objet est connu par le développeur et n'a pas de raison de changer au cours de l'exécution du programme – ou changera, mais restera dans la même hiérarchie de classe. Il est alors préférable d'utiliser le typage statique et de disposer ainsi de l'aide du compilateur afin réduire les risques de bogues.

Toutefois, il est possible que la référence soit amenée à pointer vers différentes classes, il faudra alors utiliser le typage dynamique. Auquel cas le compilateur ne pourra pas réaliser l'ensemble des vérifications qu'il fait lors du typage statique, et vous serez exposé à des erreurs lors de l'exécution du code (ce que l'on appelle un crash).

Le rôle du pointeur `isa`

Comme nous l'avons vu dans les deux sections précédentes, les objets sont typés de manière dynamique. Ceci signifie que le compilateur ne sait pas

forcément à quel type d'objet il a affaire. Il faut donc fournir à l'environnement d'exécution un moyen d'obtenir l'information. C'est là qu'intervient `isa` : par définition, chaque objet pointe *via* `isa` vers son objet-classe.

Toutefois, il est important de noter que vous n'emploierez jamais le pointeur `isa` directement. Il est utilisé exclusivement par l'environnement d'exécution. En revanche, vous pourrez envoyer le message `class` (voir section "Envoyer un message") à tout objet afin d'obtenir son objet-classe, du type `Class` :

```
ClasseGuideDeSurvie * instance =
➥ [[ClasseGuideDeSurvie alloc] init];
id MaClasse = [instance class];
Class ToujourMaClasse = [instance class];
NSLog(@"MaClasse : %@",  MaClasse);
NSLog(@"ToujoursMaClasse : %@",  ToujourMaClasse);
```

Vous devriez voir sur la console le résultat suivant :

```
GuideDeSurvie[6981:10b] MaClasse :
➥ ClasseGuideDeSurvie
GuideDeSurvie[6981:10b] ToujoursMaClasse :
➥ ClasseGuideDeSurvie
```

Veuillez noter qu'il est possible d'utiliser le type `id` ou `Class`, mais qu'il n'est pas possible d'utiliser le nom de la classe, puisque cette syntaxe est utilisée pour typer la définition des instances de ces mêmes

classes. Voici un exemple pour clarifier. La syntaxe de la Figure 1.2 est incorrecte :

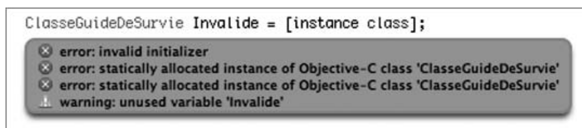


Figure 1.2 : Le compilateur génére une erreur si vous essayer d'utiliser le nom de la classe comme type de retour du message `class`.

La classe racine

Une classe racine est une classe ne possédant aucun parent dans la hiérarchie des classes et dont dérivent (directement ou indirectement) toutes les autres classes.

L'implémentation de l'environnement d'exécution d'Objective-C et des bibliothèques de base fournies par Apple contient la définition de la classe `NSObject` comme classe racine.

`NSObject` est donc l'équivalent en Objective-C de `Java.lang.Object` ou de `Object` en C#. Toute classe hérite de `NSObject`, ce qui est une bonne chose car vous n'avez ainsi pas à vous préoccuper d'écrire tout le code nécessaire à la bonne intégration de vos classes dans l'environnement d'exécution d'Objective-C. Il est en théorie possible d'écrire sa propre classe racine, mais en pratique, c'est une tâche extrêmement difficile, périlleuse et – soyons réaliste – inutile.

`NSObject` fournit une implémentation minimale de plusieurs méthodes que vous serez amené à utiliser très souvent, notamment :

```
+ (void)initialize;
- (id)init;
+ (id)new;
+ (id)alloc;
- (void)dealloc;
- (id)copy;
- (id)mutableCopy;
+ (NSString *)description;
```

Pour aller plus loin

Vous avez sans doute déjà bien compris l'idée principale de cette section : toutes les classes en Objective-C sur plateforme Apple dérivent de `NSObject`, qui est la classe racine.

En fait, ce n'est pas tout à fait exact. Comme nous l'avons vu, il est possible de définir plusieurs classes racines et, en de très rares occasions, ceci devient même une nécessité. C'est le cas par exemple de la classe `NSProxy`, qui est une classe racine, mais qui implémente le protocole `NSObject`.

La raison est tout à fait logique : le rôle de `NSProxy` est de fournir l'infrastructure nécessaire à la création d'objets distants (ainsi que d'autres utilisations en relation avec les objets distants et les systèmes distribués).

Différence entre id et NSObject

```
id objetDeclareAvecId; // cas 1
NSObject *objetDeclareAvecNSObject; // cas 2
```

Étant donné que `id` est un pointeur vers un objet et que toutes les classes en Objective-C héritent de `NSObject`, il est facile d'arriver à des conclusions trop hâtives. Nous allons donc dissiper toute confusion et profiter de cette opportunité pour faire un point intéressant et récapituler les cinq sections précédentes.

Prenons donc les deux cas précédents :

- Le cas 1 est le cas le plus courant. Nous avons vu précédemment que nous déclarons ici simplement un objet sans préciser son type. Le compilateur ne sait pas à quel type d'objet il est confronté. De plus, comme le typage dynamique le permet, il autorise l'envoi de n'importe quel message sans essayer d'appliquer la moindre contrainte. Les erreurs seront découvertes lors de l'exécution par l'environnement d'exécution.
- Dans le cas 2, nous déclarons un pointeur vers un objet de type `NSObject`. Il y a tout d'abord une petite différence syntaxique : nous utilisons l'étoile (`*`) qui n'est pas nécessaire avec `id`. De plus, ici, l'objet est nécessairement de type `NSObject`. Il est peut-être intéressant de rappeler ici que tous les objets n'héritent pas nécessairement de `NSObject` (voir la note "Pour aller plus

loin” de la section précédente). Cette situation ne devrait se produire que très rarement et dans des cas particuliers que vous aurez soigneusement délimités. Mais, ici, le compilateur va procéder à une analyse statique des messages envoyés à l'objet, et générer un avertissement pour chaque message auquel `NSObject` ne répond pas.

En conclusion, le cas 2 est celui utilisé par les développeurs Java et C# par exemple, mais étant donné qu'Objective-C dispose d'une syntaxe dédiée pour le typage dynamique, c'est cette syntaxe qui devra être utilisée dans notre code Objective-C.

`nil`, `Nil` et `NULL`

Le mot-clé `nil` désigne un objet nul, tandis que le mot-clé `Nil` désigne un objet-classe nul. La capitalisation de la première lettre permet de différencier l'instance nulle d'une classe de l'objet-classe nul. Il est intéressant de noter qu'il est équivalent de dire qu'un objet (ou un objet-classe) est nul ou que c'est une référence vers `nil` ou que la valeur du pointeur `id` est 0. En effet, il est alors possible de tester la valeur des pointeurs pour s'assurer que l'objet n'est pas nul et donc qu'il est possible de lui envoyer un message (d'invoquer une méthode).

`NULL` est une définition utilisée dans les API C/C++ et souvent précisée par une directive compilateur. Il n'est pas utilisé en Objective-C, sauf lors de l'utilisation de bibliothèques écrites en C ou C++.

Comme nous l'avons vu précédemment, une classe Objective-C n'est autre qu'un type particulier d'objet appelé objet-classe.

`nil` et `Nil` sont tous deux définis dans le fichier `/Developer/SDKs/MacOSX10.5.sdk/usr/include/objc/objc.h` de la manière suivante :

```
#ifndef Nil
#define Nil __DARWIN_NULL /* id of Nil class */
#endif
#ifndef nil
#define nil __DARWIN_NULL /* id of Nil instance */
#endif
```

D'après cette définition, il est clair que `Nil` désigne un objet-classe nul, alors que `nil` désigne un instance de classe `NULL`. Vous pouvez aussi remarquer que `nil` et `Nil` représentent deux concepts distincts, mais que derrière les concepts se cache la même valeur : `__DARWIN_NULL`.

Il est alors possible de creuser encore un peu plus et chercher ce que signifie `__DARWIN_NULL`. Sa définition se trouve dans le fichier `/Developer/SDKs/MacOSX10.5.sdk/usr/include/sys/_types.h` :

```
#define __DARWIN_NULL ((void *)0)
```

Nous savons désormais que `__DARWIN_NULL` n'est donc qu'une redéfinition du pointeur nul de C.

Envoyer un message

```
[objet-récepteur message:argument1 nomArg2:
argument2]
```

En Objective-C, pour demander à un objet de réaliser une action (c'est-à-dire exécuter une méthode de l'objet), il faut lui envoyer un message.

Si l'objet a été typé lors de la déclaration, le compilateur vérifie que l'objet répond bien au message envoyé. S'il trouve la méthode correspondante, tout va bien. S'il ne la trouve pas parmi le jeu de méthodes de l'objet, il émet simplement un avertissement, et continue son travail. Contrairement à Java ou C# (où la liaison est statique, lors de la compilation), ici la compilation se termine sans erreur et vous pouvez exécuter le code. Ce n'est qu'à l'exécution que l'erreur sera visible (on parle alors de *liaison dynamique*).

Si l'objet n'est pas typé (déclaré avec `id`) aucune vérification n'est effectuée et le code compile. Encore une fois, si lors de l'exécution l'objet ne répond pas au message, une erreur surviendra.

Lors de l'envoi d'un message, l'environnement d'exécution parcourt l'ensemble des méthodes disponibles dans le répertoire de l'objet destinataire et s'il trouve la méthode correspondante au message envoyé, l'invoque avec les paramètres passés avec le message.

L'envoi de message correspond donc à un appel de fonctions, mais dans un environnement dynamique. La syntaxe utilisée provient de SmallTalk : `[récepteur message]`.

Voici par exemple un message `date` envoyé à l'objet-classe `NSDate` :

```
[NSDate date];
```

Pour passer des arguments à la méthode, la syntaxe devient un peu plus compliquée car il faut distinguer deux cas : les méthodes avec un seul argument et celles avec plusieurs arguments.

Dans le cas des méthodes avec un paramètre unique, le message contient le symbole "`:`" suivi de l'argument. Par exemple, pour envoyer le message `numberWithInt` à `NSNumber` avec l'argument 5, la syntaxe sera la suivante :

```
[NSNumber numberWithInt:5];
```

Lorsque les méthodes ont plusieurs arguments, un envoi de message devient :

```
NSURL* pejvanHome = [NSURL fileURLWithPath:@"
➥ /Users/pejvan" isDirectory:YES];
```

La méthode précédente de `NSURL` a pour signature `fileURLWithPath:isDirectory:`. Ceci est un peu perturbant pour les développeurs habitués aux langages

les plus courants, tels que C, C++, C# et Java. Mais, on s'y habitue très rapidement et la pratique rend totalement transparente cette différence.

De plus, cette syntaxe rend le code Objective-C très lisible car l'envoi des messages indique très clairement le rôle de chaque paramètre, et le nom de chaque paramètre additionnel préfixe l'objet passé en paramètre. Par exemple dans l'exemple précédent, `isDirectory: (nom du paramètre)` précède `YES (valeur passée)`. La signification de `YES` est donc évidente.

Info

Objective-C définit les valeurs booléennes `YES` et `NO`. La plupart des autres langages définissent `true` et `false` (C++, Java, C# par exemple) ou True et False (Python).

Objective-C étant un sur-ensemble de C et compatible avec C++ dans une large mesure, l'utilisation de `true` et `false` ne génère pas d'erreur de compilation, mais reste fortement déconseillée.

Dans un langage comme Java ou C#, la même méthode ressemblerait à :

```
//exemple hypothétique en Java ou C#
NSURL pejvanHome = NSURL.fileURLWithPath("/Users/
➥pejvan", true);
```

Il n'est pas possible dans les autres langages de savoir, par simple lecture du code, ce que signifient les paramètres additionnels. Il faut obtenir l'aide de l'environnement de développement, ou pire, se référer à la documentation.

Pour aller plus loin

Contrairement à Java ou C#, où invoquer une méthode sur un objet nul engendre une exception, il est valide d'envoyer un message à `nil` en Objective-C. Comme on peut s'y attendre, envoyer un message à `nil` n'engendre aucune action et la méthode retourne, en général, 0 (`nil`). Ceci rend la programmation défensive[3] plus simple et est relativement facilement exploitable.

Pour de plus amples informations, vous pouvez vous reporter à la section "Sending Messages to nil" de *Objective-C Programming Guide* (http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/Articles/ocObjectsClasses.html#//apple_ref/doc/uid/TP30001163-CH11-SW7).

3. Pour une description du concept deprogrammation défensive, reportez-vous à cet article du Journal du Net : http://www.journaldunet.com/developpeur/tutoriel/theo/070831-programmation-defensive.shtml ou, en anglais un article bien plus complet sur Wikipedia : http://en.wikipedia.org/wiki/Defensive_programming.

Les pointeurs `self` et `super`

```
-(id) init {
    self = [super init];
    if ( self ) {
        //procéder au reste de l'initialisation
    }
    return self;
}
```

Objective-C fournit deux mot-clés, `self` et `super`, dont le fonctionnement est semblable aux `this` et `super` respectivement de C# et Java.

Le mot-clé `self` désigne l'objet courant dans la définition d'une méthode. Donc, un message envoyé à `self` est un message envoyé à l'objet courant (self signifiant *soi-même*, en anglais), tandis que `super` désigne la classe parente. Ainsi, un message envoyé à `super` va être résolu par l'environnement d'exécution en appelant l'implémentation de la méthode telle que définie dans la classe parente (récursivement si la classe parente ne définit pas la méthode).

Les mot-clés `self` et `super` sont très utilisés en Objective-C, notamment pour l'allocation et l'initialisation des instances.Vous serez donc amené à les rencontrer très fréquemment.

Définir une classe

```
@interface ClasseGuideDeSurvie : ClasseParente {
    //déclaration des membres de la classe se
    //fait ici
}
//déclaration des méthodes de classe et
//d'instance se fait ici
- (id) methodeDeMembre;
- (NSString *) methodeUnParametre: (NSString *)
  ➥ parametre1;
- (void) methodeAvecDeuxParametres: (NSString *)
  ➥ parametre1 secondParametre:(NSString
*)parametre2;
+ (id) methodeDeClasse;
@end
```

Cette déclaration de classe est typique.

La déclaration et l'implémentation des classes en Objective-C sont proches de la manière de faire en C++ dans le sens où la déclaration se fait dans un fichier d'en-tête (extension `.h`), tandis que l'implémentation se trouve dans un fichier implémentation (extension `.m`) et diffère de l'approche mono-fichier de Java et C#.

La convention veut donc que chaque classe soit définie par ses fichiers d'en-tête et d'implémentation (avec quelques variations possibles que nous verrons par la suite) et qu'une et une seule classe soit définie dans chaque fichier. Ces fichiers portent alors le nom de la classe (par exemple : `ClasseGuideDeSurvie.h` et `ClasseGuideDeSurvie.m`).

Deux mot-clés sont importants à retenir ici : `@interface` et `@implementation`, qui, comme vous l'avez deviné, représentent la déclaration de l'interface et de l'implémentation.

Comme nous l'avons vu précédemment, toute classe Objective-C dérive de la classe racine qui, dans le cas le plus courant, est `NSObject`. Vous n'êtes donc pas obligé de le spécifier dans votre déclaration, mais il est recommandé de le faire afin de ne pas recevoir d'avertissements de la part du compilateur.

Voici à quoi ressemble la déclaration d'une classe minimaliste, dérivant simplement de `NSObject` :

```
@interface ClasseGuideDeSurvie : NSObject {
}
```

Comme vous l'avez compris sur l'exemple précédent, pour dériver une classe d'une autre classe, il faut faire suivre la classe dérivée de ":" et du nom de la classe parente.

À retenir que les membres sont déclarés entre les accolades (voir section "Encapsuler les données internes aux classes" pour davantage d'informations) et que les méthodes sont déclarées après l'accolade fermante. Les méthodes de membre sont précédées du signe moins (-) tandis que les méthodes de classe sont précédées du signe plus (+).

Il ne reste plus qu'à créer le fichier d'implémentation, dont le contenu sera de la forme :

```
#import "ClasseGuideDeSurvie.h"
@implementation ClasseGuideDeSurvie
- (id) methodeDeMembre {
    //implémentation
}
- (NSString *) methodeUnParametre: (NSString *)
   ➥ parametre1 {
    //implémentation
}
- (void) methodeAvecDeuxParametres: (NSString *)
   ➥ parametre1 secondParametre:(NSString
*)parametre2 {
    //implémentation
}
+ (id) methodeDeClasse {
    //implémentation
}
@end
```

Un fichier d'implémentation importe normalement le fichier d'en-tête (avec la directive `#import`, voir section suivante à ce sujet) avant de fournir l'implémentation des méthodes entre les directives `@implementation` et `@end`.

La directive #import

```
#import <Foundation/Foundation.h>
//en-tête système
#import "ClasseGuideDeSurvie.h" //en-tête projet
```

Cette directive importe les définitions contenues dans le fichier d'en-tête cible afin que le compilateur puisse résoudre les références. `#import` est typiquement utilisé pour importer les définitions contenues dans les frameworks du système ainsi que les en-têtes des classes propres à vos projets.

L'importation des en-têtes du système (se trouvant dans le *path* du compilateur) se fait en encadrant le nom du fichier par < et > :

```
#import <Foundation/Foundation.h>
```

L'importation des fichiers propres au projet se fait en mettant le nom du fichier entre guillemets :

```
#import "ClasseGuideDeSurvie.h"
```

La directive `#import` est similaire à son équivalent Java, C# ou Python et est, en général, connue de tous les développeurs. Elle s'assure que chaque fichier est importé une seule fois. Les développeurs C et C++ sont davantage familiers avec la directive `#include` (et les `#if` et `#endif` qu'elle engendre en raison du risque d'importation multiple).

`#import` est donc une directive importante et très courante, puisque chaque fichier source contient en général une ou plusieurs directives `#import`.

Une directive proche d'`#import` est la directive `@class` que nous allons voir dans la section suivante.

La directive @class

```
@class NomDeClasse
```

À la différence de la directive `#import`, qui importe des définitions depuis un autre fichier, la directive `@class` sert juste à signaler au compilateur l'existence d'une certaine classe sans pour autant introduire sa définition ou son interface.

Cette directive est en général utilisée dans deux principaux cas :

- Déclarer l'existence d'une certaine classe dans le fichier d'en-tête de la classe que vous êtes en train de définir. Apple explique que cela permet de minimiser le code vu par le compilateur, de réduire ainsi le nombre de fichiers importés au moment de la compilation et d'éviter les erreurs que cela pourrait engendrer tout en simplifiant le code. Il est toutefois toujours nécessaire d'importer l'en-tête dans le fichier d'implémentation de votre classe.

- Permettre la définition de deux classes interdépendantes. Un exemple rendra ce cas beaucoup plus clair.

  Il serait impossible de définir et de compiler les fichiers `A.h`, `A.m`, `B.h` et `B.m` sans `@class` :

Contenu de `A.h` :

```
@class B;
@interface A : NSObject {
    B * membre1;
    B * membre2;
}
@end
```

Contenu de `B.h` :

```
@class A;
@interface B : NSObject {
    A * membre1;
    A * membre2;
}
@end
```

`@class` permet ainsi la définition de classes interdépendantes et évite les problèmes liés aux définitions circulaires.

Déclarer un protocole formel

```
@protocol MonProtocole
- methode1:(NSStream *) stream;
- methode2:(NSCalendarDate*) date
➥ vers:(NSString*)   maChaine ;
@end
```

Objective-C dérive de C où le mot interface désignait le fichier d'en-tête. La directive compilateur `@interface` est utilisée pour introduire la définition d'une classe. Objective-C appelle donc protocole formel, le concept que les développeurs Java et C# connaissent sous le nom d'interface. Nous verrons qu'Objective-C définit aussi le concept de protocole informel.

Un protocole permet de définir un ensemble de méthodes qui sera alors utilisé par différentes classes (et qui devront donc implémenter ces méthodes). On dit alors que ladite classe implémente ce protocole. Le protocole ne fournit donc pas d'implémentation, mais juste la définition des méthodes.

C'est un concept simple, mais très puissant, qui permet de résoudre la majorité des problèmes liés à l'héritage multiple de C++ (et Python dans une moindre mesure).

Il est très simple de déclarer un protocole : il suffit d'utiliser la directive compilateur `@protocol` suivie du nom du protocole, de lister les signatures des méthodes du protocole et de terminer par `@end`.

Par défaut, l'implémentation de toutes les méthodes déclarées dans un protocole formel devient obligatoire pour la classe qui décide d'implémenter ce protocole.

Par convention, les protocoles sont déclarés dans un fichier d'en-tête (extension `.h`) portant le nom du protocole (ici `MonProtocole.h`).

Rendre optionnelles certaines des méthodes d'un protocole

```
@protocol MonProtocole
@optional
- methode1:(NSStream *) stream;

@required
- methode2:(NSCalendarDate*) date
➥ vers:(NSString*) maChaine ;
@end
```

Par défaut, lorsqu'une classe implémente un protocole formel, elle doit implémenter toutes les méthodes du protocole et le compilateur est là pour s'assurer que c'est bien le cas. Il est toutefois possible de rendre facultatives certaines méthodes d'un protocole grâce à la directive compilateur `@optional`. La directive `@required` est donc la valeur par défaut.

Voici le protocole de l'exemple de la section précédente (section "Déclarer un protocole formel"),

mais avec la `methode1` devenue optionnelle et la `methode2` devenue explicitement requise.

Bien qu'il est possible de rendre certaines méthodes optionnelles, nous vous déconseillons de suivre cette approche. En effet, votre code devient beaucoup plus compliqué à gérer dès que vous introduisez des méthodes facultatives étant donné qu'il n'est pas possible de savoir à l'avance si une instance implémente la méthode optionnelle, vous devez effectuer des vérifications avant chaque appel.

Une manière élégante de résoudre ce problème consiste à séparer de manière sensible le protocole en séparant les méthodes optionnelles des méthodes obligatoires. Ainsi, la classe qui ne souhaitait pas implémenter les méthodes optionnelles n'implémente qu'un protocole, celui des méthodes obligatoires. Une autre classe, qui implémentait les méthodes optionnelles, implémente désormais deux protocoles différents.

L'exemple précédent deviendrait alors :

```
@protocol MonProtocoleFacultatif
- methode1:(NSStream *) stream;
@end
@protocol MonProtocoleObligatoire
- methode2:(NSCalendarDate*) date
➥ vers:(NSString*) maChaine ;
@end
```

Créer un protocole informel

```
@interface NSObject (MonProtocole) // ou
➥ (NSObject)
- methode1:(NSStream *) stream;
- methode2:(NSCalendarDate*) date
➥ vers:(NSString*) maChaine ;
@end
```

Info

Cette section requiert la connaissance du concept de Categorie, défini à la section "Les catégories" de ce chapitre.

Un protocole informel consiste à définir un ensemble de méthodes dans une catégorie de la classe parente à une hiérarchie de classes spécifiques. Il est également possible de l'appliquer directement à `NSObject` pour la rendre totalement générale. En effet, comme toutes les classes dérivent directement ou indirectement de `NSObject`, cette catégorie devient totalement générique et il devient possible de l'appliquer à n'importe quelle classe. Mais à la différence des catégories normales, aucune définition des méthodes n'est fournie.

Une fois l'interface de la catégorie définie, il faut encore redéclarer les méthodes dans l'interface des classes qui vont implémenter ce protocole informel avant de les définir dans le fichier d'implémentation.

Un protocole informel est donc à cheval entre protocole et catégorie puisque d'un côté, il est défini comme étant une catégorie, mais sans implémentation, et d'un autre côté, ce n'est pas un protocole au sens propre puisqu'il n'y a pas de vérification effectuée par le compilateur et toutes les méthodes sont optionnelles.

Je présente le protocole informel à titre informatif et pour vous faire découvrir la fonctionnalité à titre de curiosité. En pratique, je conseille de les éviter et d'utiliser des protocoles formels avec des méthodes requises autant que possible.

Info

`@protocol` s'utilise également de la même manière que `@class` (voir section "La directive @class") afin de permettre la déclaration de protocoles interdépendants et d'éviter les problèmes liés aux définitions circulaires.

Adopter un protocole

```
#import <Foundation/Foundation.h>
#import "MonProtocole.h"
@interface ClasseGuideDeSurvie : NSObject
➥<MonProtocole, MonSecondProtocole> {
     //déclarations…
}
- exemple1;
- (id) exemple2;
- (void) exemple3;
@end
```

La syntaxe pour adopter un protocole est très simple. À la différence de C# où adopter une interface ou dériver d'une classe parente se fait exactement avec la même syntaxe, Objective-C propose une syntaxe différente pour chaque cas. Il est donc plus proche, en ce sens, de Java qui différencie explicitement la dérivation (mot-clé `extends`) et l'implémentation d'une interface (mot-clé `implements`).

Une classe adopte un (ou plusieurs) protocole(s) en le(s) listant entre les signes `<` et `>` après la classe parente lors de la déclaration. Il est fort heureusement possible pour une classe d'adopter plusieurs protocoles. Il faut alors séparer les protocoles par une virgule.

Info

Il ne faut pas oublier d'importer le fichier d'en-tête déclarant le protocole (`MonProtocole.h` ici).

Les protocoles, comme les interfaces en C# et Java, peuvent également adopter des protocoles. Nous avons alors des protocoles imbriqués. Il n'y a rien de difficile et la syntaxe est tout à fait logique.

Par exemple, si `MonProtocole` devait implémenter le protocole `NSCoding`, nous aurions :

```
#import <Foundation/Foundation.h>

@protocol MonProtocole <NSCoding>
@optional
```

```
- methode1:(NSStream *) stream;
@required
- methode2:(NSCalendarDate*) date
vers:(NSString*) maChaine ;
@end
```

Toute classe adoptant `MonProtocole` doit alors adopter également le protocole `NSCoding` soit en héritant les méthodes soit en les implémentant directement.

Encapsuler les données internes aux classes

```
#import <Foundation/Foundation.h>
#import "MonProtocole.h"
@interface ClasseGuideDeSurvie : NSObject
➥<MonProtocole> {
    @public
    id membrePublic;
    @protected
    id membreProtected;
    id autreMembreProtected;
    @private
    id membrePrivate;
    @package
    id membrePackage;
}
- exemple1;
- (id) exemple2;
- (void) exemple3;
@end
```

Similairement à Java et C#, Objective-C permet de limiter la visibilité des variables d'instance avec des directives du compilateur `@public`, `@protected` et `@ private` (et aussi `@package` avec Objective-C 2.0 et en mode 64 bits uniquement).

La définition de ces directives correspond exactement à celle de leurs homologues en C++, C# et Java. Du plus restrictif au moins restrictif, nous avons :

- `@private`. La variable d'instance n'est accessible que depuis le code source de la classe qui la définit.
- `@protected`. La variable d'instance n'est accessible que depuis le code source de la classe qui la définit et des classes qui dérivent de cette classe.
- `@public`. La variable d'instance est visible depuis n'importe quel source code.
- `@package`. La variable d'instance est considérée publique à l'intérieur de la bibliothèque où elle est définie, mais comme privée à l'extérieur (nouveauté d'Objective-C 2.0). Cette visibilité est similaire aux `internal` de C# et `package private` de Java.

La valeur par défaut, lorsqu'aucune directive n'est donnée est `@protected`. Ce comportement est donc différent de C# , C++ (`private`) et Java (`package internal`).

Comme vous l'avez sans doute remarqué ici, nous n'avons pas spécifié de visibilité pour les méthodes de notre classe. En effet, contrairement à C++, C# et Java, Objective-C ne dispose pas de directive pour limiter la visibilité des méthodes. Ceci peut sembler assez limitant au début, mais, en fait, on s'y habitue rapidement. De plus, il existe une convention à connaître et à respecter que nous allons voir dans la section suivante.

Déclarer une méthode protégée ou privée

Objective-C ne propose pas de moyen de créer des méthodes protégées ou privées.

Toutefois, Objective-C, dérivant directement de C, fonctionne encore avec le principe de fichier d'en-tête (extension `.h`) et fichier d'implémentation (extension `.m` équivalent à `.cpp` en C++). De plus, le langage étant dynamique, certaines lois propres aux langages statiques ne s'appliquent pas.

Par exemple, il est possible de déclarer une méthode dans l'interface d'une classe, mais de ne pas fournir la définition dans le fichier d'implémentation. Le compilateur vous prévient alors qu'il ne trouve pas l'implémentation de ladite méthode, mais votre code est compilé et prêt à exécuter. C'est seulement lors de l'exécution que l'environnement fait la liaison dynamique et qu'une erreur survient si l'implémentation manque.

En se basant sur la liaison dynamique, on comprend qu'il est également possible de fournir la définition d'une méthode dans le fichier d'implémentation sans pour autant l'avoir déclarée dans le fichier d'en-tête.

Une fois le code compilé, c'est le fichier d'en-tête que vous fournissez à vos utilisateurs, ainsi que le code compilé. L'existence même de cette méthode est alors cachée à l'utilisateur de votre classe.

Cette convention est utilisée pour les méthodes privées, mais vous remarquerez que ni le compilateur ni l'environnement d'exécution ne mettent de barrières physiques à l'utilisation d'une méthode ainsi cachée. Il ne s'agit donc pas à proprement parler d'une méthode protégée ou privée, mais plutôt d'une méthode masquée.

Objective-C 2.0 introduit un autre moyen d'obtenir un résultat similaire avec les extensions (voir section "Les extensions").

Instancier une classe

```
ClasseGuideDeSurvie * instance =
➥ [[ClasseGuideDeSurvie alloc] init];
```

Une instance est initialisée en envoyant de manière imbriquée les messages `alloc` et `init` à l'objet-classe correspondant.

Nous fournissons ici une réponse superficielle à la question. Un chapitre entier est nécessaire pour bien comprendre le fonctionnement de la gestion de la mémoire en Objective-C. Nous reportons donc ces explications au Chapitre 2, qui est entièrement dédié à la gestion de la mémoire.

Initialiser un objet-classe

```
// static int nombreDInstantiations; dans le
   ➥ fichier d'en-tête
#import "ClasseAvecCompteurDInstances.h"
@implementation ClasseAvecCompteurDInstances
-(id) init {
    self = [super init];
    nombreDInstantiations ++;
    return self;
}
+(int) nombreDinstantiation {
    return nombreDInstantiations;
}
+(void) initialize {
    nombreDInstantiations = 0;
}
@end
```

Il est parfois nécessaire d'avoir accès au constructeur d'un objet-classe afin, par exemple, d'initialiser des variables statiques. C# dispose de constructeurs statiques et Java de blocs statiques. Objective-C propose la méthode de classe `initialize`.

Tout comme C# et Java, c'est l'environnement d'exécution d'Objective-C qui s'occupe d'appeler la méthode de classe pour vous. La méthode de classe `initialize` est appelée avant que l'objet-classe ne reçoive son tout premier message.

Voici un piège dans lequel il est très facile de tomber, mais qu'il est également facile d'éviter : si une classe n'implémente pas la méthode de classe `initialize` (la majorité des classes n'a pas besoin de l'implémenter), l'environnement d'exécution va appeler la méthode `initialize` de la classe parente. En effet, la classe dérivée hérite automatiquement de la méthode `initialize` de sa classe parente. Cela signifie que cette méthode `initiliaze` sera appelée deux fois (ou plus). Ceci peut avoir des conséquences désastreuses (plantage de votre application) car l'état de votre classe n'est plus cohérente.

Heureusement, il est très facile d'éviter ce piège. Il suffit de vous assurer que la classe n'est pas déjà initialisée dans votre méthode `initialize` avant de procéder :

```
+ (void) initialize{
     //vérifier que l'appel ne remonte pas
     //d'une classe fille
     if (self == [ClasseGuideDeSurvie class]){
                  //procéder à l'initialisation
     }
}
```

Les catégories

```
#import "ClasseGuideDeSurvie.h"
@interface ClasseGuideDeSurvie
➥(NomDeMaCategorie)
// declaration des méthodes de la catégorie
@end
```

Les catégories (et les extensions) permettent d'ajouter des méthodes à des classes existantes. Elles permettent ainsi d'étendre les fonctionnalités d'une classe sans avoir besoin de définir une sous-classe (c'est-à-dire sans recourir à la dérivation). L'aspect le plus puissant des catégories réside dans leur capacité à étendre les classes sans avoir accès à leur source code. Il devient alors possible d'adapter à ses propres besoins les classes des frameworks du système et bibliothèques tierces. Nous disposons donc d'une méthode simple et élégante d'étendre les classes.

Alors que les catégories permettent d'étendre la définition des classes (comme si de nouvelles fonctions étaient déclarées dans leurs fichiers d'en-têtes), les extensions permettent de déclarer de nouvelles méthodes dont l'implémentation sera requise par le compilateur. Cette implémentation pourra alors se faire dans des fichiers d'implémentation différents, distincts du fichier d'implémentation principal.

Quelques points très importants en ce qui concerne les catégories :

- Toutes les variables d'instance de la classe sont directement accessibles par la catégorie, y compris les instances marquées privées avec `@private`.
- Les méthodes ajoutées à une classe par le biais d'une catégorie sont également héritées par les classes dérivées.
- Les catégories peuvent fournir une nouvelle implémentation d'une méthode préexistante déjà déclarée dans l'interface de la classe.
- Lors de l'exécution, il devient impossible de distinguer une méthode "pure" (déclarée dans l'interface de la classe et définie dans son fichier d'implémentation) d'une méthode ajoutée *via* une catégorie.
- Une catégorie peut également adopter un protocole.

Les catégories et extensions sont propres à Objective-C dans le sens où il n'existe pas d'équivalents en C++, C# ou Java. Récemment C#, en version 3.0, s'est vu ajouter les extensions de classe, assez similaires en termes de fonctionnalité.

Nous allons voir quelques cas d'utilisation classique des catégories et des extensions dans les sections suivantes.

Déclarer et implémenter une catégorie

```
#import "ClasseGuideDeSurvie.h"
@interface ClasseGuideDeSurvie (MaCategorie)
- (void) methodeCategorie;
+ (id) methodeDeClasseCategorie: (NSString *)
➥ parametre1 AvecSecondParametre:(NSString
➥*)parametre2;
@end
```

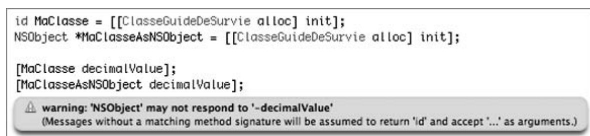
La déclaration et l'implémentation d'une catégorie sont très similaires à celles d'une classe. Il ne faut pas oublier qu'une catégorie sert à ajouter des fonctionnalités à une classe pré-existante. Ceci implique qu'il faut d'abord importer le fichier d'en-tête de la classe à "catégoriser", puis déclarer l'interface de la catégorie comme étant une interface de la classe. Mais, cette fois, il faut faire suivre le nom de la classe par le nom de la catégorie entre parenthèses.

Apple recommande de créer un fichier d'en-tête et un fichier d'implémentation suivant la nomenclature `NomClasse+NomCatégorie.h` et `NomClasse+NomCatégorie.m`.

Dans l'exemple précédent, nous avons déclaré de la catégorie `MaCategorie` pour notre classe `ClasseGuideDeSurvie`.

Comme nous l'avons vu au cours des sections précédentes, le compilateur n'émet que des avertissements s'il ne trouve pas les implémentations.

À titre d'illustration, voici ce que nous obtiendrions si nous laissions l'implémentation vide (voir Figure 1.3).


id MaClasse = [[ClasseGuideDeSurvie alloc] init];
NSObject *MaClasseAsNSObject = [[ClasseGuideDeSurvie alloc] init];

[MaClasse decimalValue];
[MaClasseAsNSObject decimalValue];
warning: 'NSObject' may not respond to '-decimalValue'
(Messages without a matching method signature will be assumed to return 'id' and accept '...' as arguments.)

Figure 1.3 : Une méthode non implémentée ne gère qu'un avertissement du compilateur, pas une erreur.

Info

Il est important de bien lire les avertissements du compilateur. En effet, contrairement à C# ou Java, les avertissements sont extrêmement importants puisqu'une erreur grossière comme une implémentation manquante est signalée simplement avec un avertissement.

Enfin, voici à quoi pourrait ressembler le fichier d'implémentation de notre catégorie `ClasseGuideDeSurvie+MaCategorie.m` :

```
#import "ClasseGuideDeSurvie+MaCategorie.h"
 @implementation ClasseGuideDeSurvie (MaCategorie)
 - (void) methodeCategorie {
     NSLog(@"methode de categorie appelée");
 }
 + (id) methodeDeClasseCategorie: (NSString *)
 ➥ parametre1 AvecSecondParametre:(NSString
 ➥*)parametre2 {
```

```
    NSLog(@"parametre1: %@ - parametre2: %@",
    ➥ parametre1, parametre2);
    return self;
}
@end
```

Vous pouvez alors compiler le code source, importer l'en-tête de la catégorie dans votre projet et l'utiliser comme si les méthodes contenues avaient été directement implémentées par la classe.

Étendre une classe sans avoir accès à son code source

```
#import <Foundation/Foundation.h>
@interface NSString (MaSecondeCategorie)
- (void) MethodePourNSString;
@end

#import "NSString+MaSecondeCategorie.h"
@implementation NSString (MaSecondeCategorie)
- (void) MethodePourNSString{
    NSLog(@"MethodePourNSString");
}
@end
```

Comme nous l'avons vu aux sections "Les catégories" et "Déclarer et implémenter une catégorie", il suffit de déclarer et d'implémenter une catégorie sur la classe à étendre. Prenons, par exemple, la classe

NSString de *Foundation.framework*. Étant donné que c'est une classe du système, nous ne disposons pas du source, mais nous souhaitons y ajouter une méthode utile pour notre projet, mais sans grand intérêt pour Apple.

L'exemple précédent montre le contenu des fichiers NSString+MaCategorie.h et NSString+MaCategorie.m servant à étendre la classe NSString.

Les extensions

```
@interface ClasseGuideDeSurvie ( )
//déclarations de méthode
@end
```

Les extensions ne sont qu'une déclinaison des catégories et leur sont donc très similaires. Une extension est en fait une catégorie sans nom, introduite par des parenthèses vides. Les extensions sont donc des *catégories anonymes*.

Info

Les extensions de classe sont une nouveauté d'Objective-C 2.0. Étant donné qu'elles ne sont qu'une variation des catégories et par souci de cohérence avec le reste du texte, nous les avons placées dans ce chapitre et non pas au Chapitre 3.

Mais en fait, les similarités s'arrêtent ici. En effet, contrairement aux catégories, les extensions introduisent des méthodes dont l'implémentation doit obligatoirement se faire dans le fichier d'implémentation de la classe et non dans un fichier séparé comme pour l'implémentation des catégories. Cette contrainte implique donc qu'il faut disposer du source code de la classe afin de pouvoir lui appliquer une extension.

De plus, contrairement au cas des catégories, Apple ne propose pas de convention de nommage pour les fichiers d'en-tête déclarant les extensions.

Nous allons voir le principal intérêt des extensions dans la section suivante.

Déclarer une méthode comme privée avec les extensions

```
@interface ClasseGuideDeSurvie ( )
- (void) methodePrivee;
@end
```

Nous avons vu à la section "Encapsuler les données internes aux classes" qu'il est impossible de déclarer des méthodes comme protégées ou privées en Objective-C et qu'il faut recourir à des artifices, des astuces et des conventions pour simuler le comportement souhaité et les extensions se prêtent très bien à ce petit jeu.

L'exemple ci-dessus montre comment ajouter une méthode privée à notre classe. Il suffit de créer un fichier d'en-tête avec la déclaration de l'extension et des méthodes privées à implémenter, puis de fournir l'implémentation de ces méthodes dans le bloc `@implementation` dans le fichier `ClasseGuideDe-Survie.m` :

```
- (void) methodePrivee {
    NSLog(@"Methode privee");
}
```

Une méthode déclarée dans une extension doit être forcément implémentée dans le fichier d'implémentation principal de la classe. Il n'est pas possible de l'implémenter dans une catégorie.

Cette manière de procéder est plus propre que celle de la section "Déclarer une méthode protégée ou privée", puisque le compilateur peut vérifier au moment de la compilation que les méthodes existent et il ne vous lance pas les nombreux avertissements que vous subissiez autrement.

Couper la définition d'une classe sur plusieurs fichiers source

En reprenant l'idée des catégories et des extensions des sections précédentes, il devient clair qu'il est possible de définir différentes catégories pour une

classe, afin de scinder le code dans différents fichiers de déclaration et d'implémentation.

Parmi les avantages, on peut citer :

- La collaboration dans les équipes est alors simplifiée, puisque chaque personne peut travailler sur son propre fichier.
- Les fichiers source ont tendance à être mieux structurés car les méthodes sont organisées par groupe.
- La compilation de gros projet est plus simple et plus rapide.

Il est possible de séparer un fichier implémentation, par exemple en créant un fichier pour chaque protocole implémenté ou/et par thème ou fonctionnalité. Il n'existe pas de recette particulière si ce n'est l'intuition et le degré de confort de chaque développeur.

Utiliser le mot-clé `static`

Objective-C ne définit pas le mot-clé `static`. Donc contrairement à C++, C# et Java où `static` désigne un membre ou une méthode appartenant à la classe même, au lieu d'appartenir à une instance donnée. Étant donné que `static` n'est pas un mot-clé d'Objective-C, cela signifie qu'il est hérité du C et donc que son utilisation suit les mêmes règles que celles du langage C. Il vous est donc vivement

conseillé de vous reporter à une référence sur le langage C avant d'utiliser ce mot-clé, qui possède de multiples facettes.

Créer un sélecteur

```
SEL afficherNotificationSelector =
➥@selector(afficher Notification:);
SEL methode2ParamSel = @selector
➥ (methodeAvecDeuxParametres:
➥ secondParametre:);
```

Un sélecteur de type `SEL` est obtenu au moment de la compilation avec la directive compilateur `@selector()` qui prend en paramètre la méthode dont vous souhaitez obtenir le sélecteur. Le paramètre passé n'est pas le nom de la méthode sous forme de chaîne (type `NSString`) mais bel et bien directement la méthode.

Attention

Il faut faire particulièrement attention à passer correctement le nom des méthodes lorsque vous créez un sélecteur. Les deux points font partie du nom des méthodes, ainsi que les noms précédents les paramètres.

Par exemple, il faudra passer "`afficherNotification:`" et non pas "`afficherNotification`". De même, vous passerez `methodeAvecDeuxParametres:secondParametre:` (voir exemple ci-dessus).

Comme nous l'avons vu précédemment, Objective-C utilise le concept de message pour appeler les méthodes de manière dynamique lors de l'exécution.

Les messages sont envoyés aux objets par le biais des sélecteurs : lors de la compilation, un sélecteur est créé pour chaque méthode, sachant que si différentes classes implémentent des méthodes portant le même nom, elles seront attribuées au même sélecteur.

Les sélecteurs occupent donc en quelque sorte le même rôle que les pointeurs de fonction en C ou C++ (ou, dans une moindre mesure, aux `single-cast delegates` en C#).

Les sélecteurs sont très importants pour la gestion des notifications (de même que les `delegates` pour C#), mais ils sont également utilisés à différentes occasions dans AppKit, comme par exemple lorsque vous souhaitez afficher un dialogue sous forme de feuillet par dessus une fenêtre :

```
//un selector est utilisé pour indiquer quelle
//méthode de l'observateur doit être appelée
//par le centre de notifications :
[[NSNotificationCenter defaultCenter]
➥ addObserver:instance
➥ selector:@selector(afficherNotification:)
➥ name:@"maNotification" object:instance];
```

Enfin, `NSObject` définit de nombreuses méthodes utilisant les sélecteurs et activant ainsi des fonctionnalités dynamiques, mais facilitant également l'écriture de code multi-threadée :

- `performSelector:`
- `performSelector:withObject:`
- `performSelector:withObject:withObject:`
- `performSelector:withObject:afterDelay:`
- `performSelector:withObject:afterDelay:inModes:`
- `performSelectorOnMainThread:withObject:waitUntilDone:`
- `performSelectorOnMainThread:withObject:waitUntilDone:modes:`
- `performSelector:onThread:withObject:waitUntilDone:`
- `performSelector:onThread:withObject:waitUntilDone:modes:`
- `performSelectorInBackground:withObject:`

Pour aller plus loin

Dans la très grande majorité des cas, la création du sélecteur se fait au moment de l'écriture du code, grâce à la directive compilateur `@selector()`.

Toutefois, il est parfois utile de pouvoir créer un sélecteur en passant le nom d'une méthode lors de l'exécution. Dans ces rares cas, la fonction `NSSelectorFromString()` permet de retourner le sélecteur correspondant au nom de la méthode passée en paramètre.

2

Gestion de la mémoire

Ce chapitre détaille la partie la plus importante et la plus critique d'Objective-C : la gestion de la mémoire.

C'est ici que le développeur doit faire le plus attention, car une mauvaise gestion de la mémoire ne pardonne pas en Objective-C. C'est soit le plantage immédiat soit la fuite de mémoire.

Si Objective-C est un langage relativement simple d'accès, la gestion de la mémoire demande beaucoup d'attention. Fort heureusement, il existe un ensemble de conventions à suivre qui facilite grandement cette gestion, notamment en mode géré (*managed*).

Objective-C 2.0 introduit le concept de ramasse-miettes en plus du mode de gestion manuelle. Non seulement Objective-C 2.0 (et son environnement d'exécution) est actuellement le seul langage permettant de gérer la mémoire manuellement ou automatiquement, mais de plus, il est possible d'avoir une application programmée pour s'exécuter dans un mode ou dans l'autre suivant l'hôte. Par exemple, l'application (le même fichier binaire) peut décider de fonctionner en mode "ramasse-miettes" sur Mac OS X 10.5 et de fonctionner en gestion manuelle lorsqu'elle s'exécute sur Mac OS X 10.4.

Nous allons d'abord revoir la gestion manuelle de la mémoire avant de découvrir la gestion automatique grâce au ramasse-miettes, abordé au Chapitre 3.

Il est important de bien écrire les accesseurs (*getters* et *setters*) car ils deviennent la source de beaucoup de problèmes de fuites de mémoire ou de plantage.

Savoir les écrire en mode géré reste plus qu'utile lorsque le code est destiné à être exécuté sur les systèmes 10.4 et antérieurs, mais aussi et surtout pour iPhone, qui ne supporte pas encore le mode ramasse-miettes, même dans la version majeure 3.0 du système.

Le mode géré

Comme nous l'avons évoqué, depuis Mac OS X, version 10.5, et Objective-C 2.0, Cocoa propose deux modes de gestion de la mémoire pour

Mac OS X. Apple a donc intégré un ramasse-miettes (*Garbage collector*) à l'environnement d'exécution d'Objective-C 2.0.

Le mode géré (*managed*) signifie dans le jargon Apple que la mémoire est gérée manuellement, *via* le mécanisme appelé *comptage de références* (ou *reference counting* en anglais). Les développeurs venus du monde COM de Microsoft sauront immédiatement ce que le comptage de références signifie.

Attention au fait que la signification de *managed* dans le monde Objective-C d'Apple et dans le monde .Net est totalement opposée. En effet, alors que pour Apple, mode géré signifie une gestion manuelle, dans le monde .Net, ceci implique que la mémoire est gérée par l'environnement d'exécution. En Java, il n'est pas possible de gérer la mémoire manuellement, donc la question ne se pose pas.

Bien que le seul intérêt de la gestion manuelle sur Mac OS X soit la rétrocompatibilité avec Mac OS X 10.4 et antérieurs (ce qui présentera de moins en moins d'intérêt dans un futur relativement proche), cette méthode prend tout son intérêt pour la plateforme iPhone OS qui ne propose pas encore le mode ramasse-miettes. La seule manière d'écrire une application pour les millions d'utilisateurs iPhone est donc de programmer en mode géré.

Au cours de ce chapitre, nous allons voir les différentes techniques et conventions à utiliser pour gérer la mémoire manuellement.

Le comptage de références

La gestion manuelle de la mémoire avec Objective-C se base sur deux piliers :

- le comptage de références ;
- la convention de possession des objets.

Le comptage de références est une technique très simple à comprendre : l'environnement d'exécution associe à chaque objet un compteur. Ce compteur représente le nombre de références pointant vers cet objet. Ce compteur doit être incrémenté à chaque fois qu'un autre objet crée une référence pour utiliser cet objet (typiquement *via* un pointeur vers l'objet). Réciproquement, le compteur est décrémenté à chaque fois qu'une référence est détruite. Une fois que l'objet n'est plus utilisé, *si tout s'est bien passé*, son compteur tombe à zéro et la mémoire allouée peut-être libérée.

Comme vous vous en êtes peut-être rendu compte, le comptage de références semble bien plus simple à dire qu'à faire ! En effet, si un objet oublie d'incrémenter le compteur, la punition est immédiate car la zone mémoire peut se retrouver libérée à tout moment et l'application plante. Si au contraire, l'objet oublie de décrémenter un compteur, l'objet ne va jamais libérer la zone de mémoire qu'il occupe et on parle alors de fuite de mémoire.

Il existe une autre limitation au système de comptage de références : le cas des dépendances cycliques.

Fort heureusement, Apple a déjà résolu tous ces problèmes par le biais de conventions, suivies par Cocoa afin d'éviter ces difficultés – et que vous devez donc également respecter à la lettre pour ne pas avoir à en subir les conséquences désastreuses.

Nous allons voir ces conventions au cours des sections suivantes et réitérons notre message : vous *devez* suivre ces conventions à la lettre, elles ne sont pas facultatives.

Compter les références

```
ClasseGuideDeSurvie * instance =
➥ [[ClasseGuideDeSurvie alloc] init];
//utilisation de l'instance
// ...
[instance release];
```

Pour créer un nouvel objet, vous disposez de différents moyens. Le premier consiste à allouer une zone de mémoire avec la méthode `alloc`, puis à initialiser un objet de la classe avec la méthode `init`. Il y a également le clonage, ou la copie, d'un objet existant *via* les méthodes du type `copy`. Enfin, il existe les différentes méthodes de classe qui renvoient un nouvel objet créé par l'objet-classe.

Ces méthodes sont appelées méthodes *utilitaires* et sont traitées à la section suivante.

Au cœur du mécanisme de comptage de références de Cocoa se trouve une règle unique : vous avez la responsabilité de libérer les objets qui vous appartiennent. Un objet vous appartient si vous l'avez explicitement créé avec l'une des méthodes `alloc`, `new`, `copy` et dérivés ou si vous vous l'êtes approprié (voir section "S'approprier un objet"). En général, vous libérez un objet en lui envoyant le message `release` (qui signifie littéralement "relâcher"). Ce message a pour conséquence la décrémentation du compteur de références de l'objet, et, si ce compteur tombe à zéro lorsqu'il reçoit le message, l'environnement d'exécution détruit l'objet et récupère la mémoire allouée.

La réciproque immédiate de cette règle unique est que vous ne libérez pas un objet qui ne vous appartient pas.

Gérer la mémoire pour les objets retournés par les méthodes de classe

```
NSURL* pejvanHome = [NSURL fileURLWithPath:@"/
➥ Users/pejvan" isDirectory:YES];
// utilisation du pointeur pejvanHome
// ...
[pejvanHome release]; // NON!
```

De nombreuses classes proposent des méthodes utilitaires facilitant l'instanciation de nouveaux objets. Ces méthodes de classe implémentent un design pattern connu sous le nom de *factory* (usine). Elles deviennent donc des usines à instances. La règle à retenir est que les objets créés par ces usines ne vous appartiennent pas : ce n'est donc pas à vous de les libérer. La logique derrière ce fonctionnement, qui peut sembler contre intuitif, est que l'usine a créé l'objet pour vous et s'occupera donc de libérer l'objet une fois que vous n'en aurez plus besoin.

L'exemple ci-dessus est donc un contre-exemple d'utilisation. Ici, l'instance pointée par le pointeur `pejvanHome` a été créée par la méthode de classe `fileURLWithPath:isDirectory` et non pas par une méthode dont le nom commence par (ou contient) `alloc`, `new` ou `copy`. Comment est-il donc possible de libérer la mémoire d'une instance ainsi allouée ? Pour résoudre cette énigme, il faut connaître l'existence du mécanisme de libération retardée ou de libération automatique de la mémoire, grâce au message `autorelease` et à la classe `NSAutoreleasePool` que nous verrons dans les sections suivantes.

Retenez que vous n'avez pas besoin de libérer explicitement les objets créés par les méthodes utilitaires sauf si vous vous les êtes appropriés explicitement (voir section "S'appropier un objet").

Gérer la mémoire des objets retournés par référence

```
NSError *erreur;
NSString *contenu = [[NSString alloc]
➥ initWithContentsOfFile:@"/Users/pejvan/test.txt"
➥ encoding:NSUTF8StringEncoding error:&erreur];
```

Certaines rares méthodes Cocoa spécifient qu'un objet est retourné par référence. C'est typiquement le cas des méthodes retournant plusieurs variables à l'utilisateur. Comme nous le verrons au Chapitre 4, les erreurs, instances de `NSError`, sont toujours retournées par référence. Prenons la méthode suivante de la classe `NSString` :

```
- (id)initWithContentsOfFile:(NSString *)path
➥ encoding:(NSStringEncoding)enc
error:(NSError**)error
```

Elle s'utilise de la manière indiquée dans l'exemple en-tête de section. Un pointeur de type `NSError` est passé par référence à la méthode `initWithContentsOfFile` qui va essayer de charger le contenu du fichier dans la chaîne de caractères `contenu`.

Si une erreur survient, `erreur` pointera vers l'instance de `NSError` contenant l'information nécessaire pour comprendre ce qui s'est produit. Toutefois, il est clair que nous n'avons pas alloué l'objet `NSError` nous-même. La méthode `initWithContentsOfFile`

s'en est chargée pour nous et nous la retourne par référence.

En conclusion, retenez qu'un objet retourné par référence a été initialisé par l'objet-classe. Il ne vous appartient donc pas et vous n'avez pas la responsabilité de libérer la mémoire qui lui a été allouée.

S'approprier un objet

```
id number = [NSNumber numberWithInt:5];
[number retain];
```

Votre classe dépend d'instances d'autres classes. Chacune de ces instances, utilisée à l'intérieur de votre classe, trouve son origine parmi les sources suivantes :

- Elle a été créée à l'intérieur de votre classe *via* les méthodes `alloc` et `init`. Auquel cas, vous avez le contrôle total de sa gestion.
- Elle a été créée à l'intérieur de votre classe, mais *via* l'une des méthodes utilitaires de son objet-classe. Auquel cas, vous n'êtes pas responsable de libérer sa mémoire : elle sera libérée automatiquement à un moment ultérieur, mais vous ne savez pas quand.
- Elle a été créée à l'extérieur de votre classe et a été passée par référence à votre instance qui possède donc un pointeur vers cet objet. Auquel cas, vous n'avez aucun contrôle sur son existence et l'instance peut être libérée à n'importe quel moment.

Vous l'avez donc sans doute pressenti : si vous ne savez pas quand un objet va être libéré, il y a de grandes chances que l'objet que vous pensez référencer n'existe plus. Lorsque cela se produit, votre programme plante et est arrêté par le système d'exploitation. Étant donné qu'une autre classe s'occupe de libérer les objets que vous utilisez, et que vous ne voulez pas qu'elle les libère alors que vous en avez encore besoin, vous réfléchissez et pensez immédiatement à deux solutions : si seulement il existait un moyen signaler aux autres classes que vous avez encore besoin de cette instance afin qu'elles ne le libèrent pas ou si seulement il existait un moyen de vous approprier l'instance afin que vous ayez le contrôle de sa gestion.

La solution adoptée par Objective-C et Cocoa est un mélange des deux méthodes suggérées ci-dessus : vous envoyez le message `retain` (qui signifie "retenir") à l'objet, afin d'indiquer que vous avez besoin de l'objet et donc qu'il ne doit pas être libéré. Un exemple est donné en début de section.

En conclusion : si la gestion de la mémoire d'un objet n'est pas de votre responsabilité, vous devez retenir l'objet en lui envoyant le message `retain` et vous devez alors le libérer une fois que vous n'en avez plus besoin en lui envoyant le message `release`.

Nous reviendrons plus en détail sur le fonctionnement de la gestion de la mémoire automatique avec les bassins de désallocation automatique (*autorelease pools*).

Gestion de l'appropriation par les collections d'objets

Les collections d'objets (ou *container* à objets), tels que `NSArray`, `NSS` et `NSDictionary`, etc., utilisent une technique appelée *référence faible* (*weak reference*). Le concept de référence faible est en fait très simple à comprendre. À la section précédente, nous avons vu qu'un objet peut s'approprier un autre objet (ou le retenir) en lui envoyant le message `retain`. C'est ce que l'on appelle une *référence forte* (*strong reference*). Une référence faible est le troisième cas de la section précédente : un objet a été créé à l'extérieur de votre classe et votre instance ne conserve qu'un pointeur vers cet objet – sans lui envoyer de message `retain`, (voir section "Fonctionnement du ramasse-miettes" au chapitre suivant).

Toutefois, un objet A possédant une référence faible vers objet B s'expose au problème évoqué dans la section suivante : il n'a aucun moyen de savoir si l'objet B a été libéré ou pas. Pour palier à ce problème, la convention adoptée est que l'objet B doit notifier sa libération en envoyant un message à l'objet A.

Un exemple très important est le cas de `NSNotificationCenter` qui a pour tâche de notifier un observateur enregistré de l'arrivée de certains événements. Le centre de notifications ne retient pas les objets qui s'abonnent (*via* la méthode `addObserver:selector:name:object:`). Les objets inscrits doivent penser

à annuler leur abonnement, en envoyant un message `removeObserver:` afin d'éviter que le centre de notifications n'engendre une erreur d'exécution.

Éviter les cycles d'appartenance

Un cycle d'appartenance est créé lorsqu'un objet A retient un objet B (en envoyant un message `retain`) et que l'objet B retient l'objet A. Lorsqu'un cycle est créé, il y a une fuite de mémoire en mode géré.

En effet pour que l'objet A soit libéré, il faut que son compteur de références passe à zéro. Mais, pour cela ne se produise, il faut que le compteur de B passe à zéro également. Or, A possédant une référence forte sur B, le compteur de B ne peut pas passer à zéro sans que A le libère.

Conclusion : pour résoudre le problème des cycles d'appartenance, la convention adoptée en Objective-C est qu'un parent retient son enfant (référence forte), mais que l'enfant se contente d'une référence faible vers son parent (comme les containers d'objet par exemple, voir section précédente).

Instancier une classe correctement

```
id MaClasse = [[ClasseGuideDeSurvie alloc] init];
```

Objective-C utilise un mécanisme en deux étapes pour instancier une classe :

- allocation de la mémoire avec la méthode `alloc` ;
- initialisation des membres avec la méthode `init`.

La méthode `alloc` alloue la quantité de mémoire nécessaire à votre objet. La bonne nouvelle, c'est que vous n'avez rien à faire pour la méthode d'allocation de la mémoire, puisque la structure de la classe a été définie au moment de la compilation. Donc, l'environnement d'exécution peut faire son travail sans avoir besoin de votre intervention.

Une fois la mémoire allouée, `alloc` s'occupe également d'initialiser toutes les variables d'instance avec leur valeur par défaut (c'est-à-dire zéro pour les valeurs, et `nil` pour les références) avant de renvoyer l'objet fraîchement alloué auquel vous devez envoyer le message `init`.

L'instanciation prend alors la forme suivante :

```
id MaClasse = [[ClasseGuideDeSurvie alloc] init];
```

Notez bien que l'instanciation se fait en une seule ligne, avec les deux méthodes imbriquées. Ceci est une convention additionnelle qu'il vous faudra respecter. En effet, Apple met en garde sur le fait qu'il est possible que l'implémentation de la méthode `init` de certaines classes retourne une nouvelle instance, qui est différente de celle retournée par la méthode `alloc`.

La méthode `init` est équivalente au constructeur en C++, C# et Java et doit donc s'assurer de bien appeler le constructeur de la super-classe avant d'initialiser les variables d'instance de sa propre classe. La méthode `init` est donc toujours de la forme :

```
- (id) init {
    if(self = [super init]) { //si la super-classe
    //renvoie nil
//(initialisation échouée) on ne fait rien
//(self sera donc nil)
        //initialisation des variables
        //d'instance ici
        membrePublic = [[MaClasse alloc] init];
        //[...]
    }
    return self;
}
```

Info

La méthode `alloc` alloue la mémoire dans la région mémoire par défaut. Il existe une variante de la méthode `alloc` appelée `allocWithZone:`. Elle permet de définir la zone mémoire dans laquelle l'objet va être alloué. Cela permet d'optimiser la gestion de la mémoire en faisant en sorte que les objets interdépendants soient alloués dans la même région de la mémoire.

Libérer la mémoire allouée à un objet

```
- (void) dealloc {
    //libérer les variables d'instance, puis
    //appeler la méthode dealloc de la
    //super-classe
    [super dealloc];
}
```

Comme nous l'avons vu précédemment, en mode géré l'environnement d'exécution d'Objective-C utilise un compteur de références pour savoir quand un objet doit être libéré. Vous n'avez donc pas libéré la mémoire explicitement, puisque l'opération est réalisée automatiquement. Cela ne signifie pas par pour autant que vous n'avez rien à faire. En effet, l'environnement d'exécution se contente d'exécuter pour vous la méthode `dealloc` de l'objet. Notez que l'environnement d'exécution appelle la méthode `dealloc` pour vous et que vous ne devez jamais l'appeler vous-même. Le rôle de cette méthode est de libérer les ressources utilisée par votre instance. En effet, le système ne peut pas libérer la mémoire des objets pointés automatiquement. Vous devrez donc dans 99 % des cas, fournir l'implémentation de la méthode `dealloc` pour votre classe.

Les deux cas où vous n'aurez pas à fournir d'implémentation de `dealloc` sont :

- Les variables d'instance de votre classe sont uniquement des valeurs (telles que `int`, `double`, etc). Ceci est un cas très peu probable car Objective-C utilise `NSNumber` et `NSString` au lieu d'employer ou de fournir des implémentations sous forme de valeurs (*value types*).
- Votre classe dérive d'une autre classe, mais se contente de définir des nouvelles méthodes, et non pas de nouvelles variables d'instance. Auquel cas, il faut se poser la question de savoir si vous n'auriez pas mieux fait de définir une catégorie (voir Chapitre 1, section "Les catégories"). Il est toutefois possible de vouloir distinguer plusieurs types en dérivant plusieurs classes filles de la même classe mère et sans définir de nouvelles variables d'instances.

L'implémentation de `dealloc` doit envoyer le message `release` à tous les variables d'instance de votre classe avant d'appeler la méthode `dealloc` de sa super-classe. Voici une implémentation typique de `dealloc` :

```
- (void) dealloc {
    [membrePublic release];
    [membreProtected release];
    [autreMembreProtected release];
    [membrePrivate release];
    [membrePackage release];
    [super dealloc];
}
```

Les autorelease pools

```
NSAutoreleasePool *MonPool = [[NSAutoreleasePool
➥ alloc] init];
//utilisation de l'autorelease pool
//[...]
[MonPool release];
```

Les *autorelease pools* (ou bassins de désallocation automatique) sont des objets particuliers et constituent l'une des pierres angulaires de la gestion de mémoire en Objective-C. Ils sont surtout utiles en mode géré, mais même avec le mode automatique, ils peuvent être employés pour aider le ramasse-miettes.

Comme nous l'avons vu à plusieurs reprises dans ce chapitre, vous n'avez pas la responsabilité de la gestion de mémoire des objets que vous n'avez pas directement créé *via* les méthodes `alloc`, `init` et `copy`. C'est là que les *autorelease pools* entrent en jeu.

Lorsqu'un objet reçoit le message `autorealease` à la place du message `release`, la mémoire qu'il occupe n'est pas immédiatement libérée. Au lieu de cela, l'objet est placé dans un bassin de désallocation automatique (*autorelease pool*) et sa libération est différée. L'objet est libéré en même temps que le bassin, c'est-à-dire lorsque ce dernier reçoit le message `release`.

À noter que les bassins se comportent comme une liste (`List`) et non pas comme un ensemble (`Set`)

dans le sens où un même objet peut y être placé plusieurs fois. L'objet reçoit alors le message `release` autant de fois que nécessaire pour être libéré.

Les *autorelease pools* sont créés et détruits de la même manière que n'importe quelle autre instance de classe. La création se fait avec l'envoi des messages `alloc/init` imbriqués à `NSAutoreleasePool` et la libération avec le message `release` envoyé à l'instance (voir exemple ci-dessus).

Enfin, il est important de savoir que les différentes instances de `NSAutoreleasePool` sont rangées dans une pile (*stack*). Ceci a plusieurs conséquences importantes :

- Chaque nouvelle instance de `NSAutoreleasePool` que vous créez se trouve empilée sur les précédentes. Tout objet qui reçoit le message `autorelease` se retrouve géré par le bassin en tête en pile. Donc la vie de l'objet dépend du bassin dans lequel il se trouve. Ceci a des conséquences bien heureuses, comme par exemple le fait que vous pouvez créer beaucoup de pools, et qu'en général, les pools vont être libérés dans le bon ordre sans que vous n'ayez à faire d'effort.
- En revanche, si vous envoyez le message `release` à un bassin de désallocation automatique qui ne se trouve pas en tête de pile, tous les bassins qui se trouvaient par dessus celui-ci, ainsi que tous les objets qu'ils contenaient, sont automatiquement libérés. Donc, si vous envoyez par accident le message `release` au mauvais bassin, votre programme

plante. D'un autre côté, si vous oubliez de libérer un bassin en lui envoyant le message `release`, il sera libéré lorsque l'un des pools se trouvant en dessous de lui est libéré. Donc la fuite de mémoire sera évitée, malgré une erreur de programmation.

Le même type de comportement se produit lorsqu'une exception est levée et que le thread sort du contexte d'exécution courant. Cette bascule de contexte a pour conséquence l'envoi d'un message `release` à l'*autorelease pool* qui se trouvait associé au contexte et, par effet domino, à tous les autres bassins se trouvant empilés sur ce dernier. La conséquence ici est que vous n'avez pas à explicitement libérer les *autorelease pools* dans vos gestionnaires d'exceptions, puisque ceci est effectué automatiquement.

Utiliser un autorelease pool

```
//initialisation de nombreux objets temporaires
//qui seront donc gérés par l'autorelease pool
//local
NSAutoreleasePool *poolLocal = [[NSAutoreleasePool
➥alloc] init];
for (int i = 0; i < nbFichiers; i++) {
    NSString *nomFichier = [listeFichiers
    ➥ objectAtIndex:i];
    NSFileHandle *handle = [NSFileHandle
    ➥fileHandleForReadingAtPath:[dossierTmp
    ➥stringByAppendingString: nomFichier]];
    //traitement des fichiers
}
```

```
//libération les objets temporaires (objets
//"autoreleasés" bien sûr, car instanciés par
//la méthode utilitaire)
[poolLocal release]; //équivalent à [poolLocal
➥ drain]
```

AppKit – le framework d'Apple pour construire des applications avec interface graphique – s'occupe d'instancier un bassin de désallocation automatique pour vous. En revanche, dans certaines conditions bien précises, il est souhaitable, voire nécessaire, de créer vos propres *pools* :

- Applications non basées sur l'AppKit comme par exemple les applications construits autour de *Foundation Framework* (créé *via* Command Line Tools > Foundation Tool, depuis le menu New Project… d'Xcode). Ici, il est absolument nécessaire de créer l'*autorelease pool* sous peine de subir des fuites de mémoire.
- Boucle engendrant la création de nombreux objets locaux à la boucle (c'est le cas souhaitable).
- Applications multi-threadées : les pools ne sont pas partagés par les threads et vous devez créer un nouveau *autorelease pool* pour chaque nouveau thread que vous engendrez. Notez également que cela signifie que chaque instance de `NSThread` dispose de sa propre pile d'*autorelease pools*.

Là encore, si vous ne créez pas l'*autorelease pool*, mais que vous envoyez le message `autorelease` à des objets, vous avez la garantie d'avoir une fuite de mémoire.

Foundation Framework

Ce cas est là spécialement à titre éducatif, en effet Xcode a la bonté de s'occuper d'insérer le code pour vous lors de la création du projet. Toutefois, il est intéressant d'en connaître la raison. Comme mentionné précédemment, les applications `NSFoundation` n'ont pas de `NSAutoreleasePool` par défaut. Il faut donc leur en attribuer un au lancement, le drainer, puis le libérer lorsque vous avez fini de l'utiliser. Toutefois, Xcode insère le code nécessaire dans votre fichier principal lorsque vous créez un projet *Foundation Tool*. Vous n'avez donc pas besoin de vous faire de soucis :

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool =
    ➥[[NSAutoreleasePool alloc] init];
    //insérer votre code ici
    [pool drain];
    return 0;
}
```

Vous remarquez donc à quel point il est trivial de créer et de gérer un `NSAutoreleasePool` : la classe

s'instancie comme n'importe quelle autre classe avec `alloc` et `init`. Lorsque vous voulez drainer le bassin, vous lui envoyez le message `drain`. Notez que le message `drain` aura le même résultat que le message `release` en mode géré, c'est-à-dire que `[pool drain]` se comporte exactement comme `[pool release]` en mode géré. Nous verrons la différence en mode automatique au Chapitre 3.

Boucle locale

Le cas le plus courant – et sans aucun doute le plus simple à gérer – se trouve être un besoin local. Le concept est simple : une boucle ou une fonction engendre la création de très nombreux objets locaux. Si vous leur envoyez le message `autorelease` sans avoir créé un bassin local, les objets seront envoyés vers le bassin principal de votre application. La mémoire allouée à ces objets ne sera donc libérée que lorsque le bassin principal est drainé. Ce qui signifie que les objets temporaires que vous avez créés occuperont la mémoire pendant un laps de temps bien plus long que nécessaire.

Une technique d'optimisation de la mémoire utilisée (pensez par exemple au cas de l'iPhone où la mémoire disponible est bien plus faible qu'un ordinateur) consiste alors à instancier un bassin local, à y déposer les objets et à drainer le pool dès la sortie de votre boucle.

Par exemple :

```
- (void) imprimerFichiersTemporaires{
    NSString* dossierTmp = @"/tmp";
    NSArray *listeFichiers = [[NSFileManager
    ➥ defaultManager]
    ➥ directoryContentsAtPath:dossierTmp];
    int nbFichiers = [listeFichiers count];
    NSAutoreleasePool *poolLocal =
    ➥ [[NSAutoreleasePool alloc] init];
    for (int i = 0; i < nbFichiers; i++){
        NSString *nomFichier = [listeFichiers
        ➥ objectAtIndex:i];
       //un "file handle" est général un objet
       //lourd qu'il est souhaitable de libérer
       //aussi rapidement que possible
       //ici, le handle est alloué via une
       //méthode utilitaire, et sa mémoire
       //est donc gérée pour nous étant
       //donné que poolLocal se trouve
       //tout en haut de la pile des
       //autoreleases pools, les NSFileHandle
       //alloués ici vont y être déposés
        NSFileHandle *handle = [NSFileHandle
        ➥ fileHandleForReadingAtPath:[dossierTmp
        ➥ stringByAppendingString: nomFichier]];
        //traitement des fichiers
        NSLog(@»traitement de %@», nomFichier);
        //suite traitement des fichiers [...]
    }
    //tous les objets ayant reçu le message
    //autorelease à l'intérieur de cette
    //fonction vont être libérés ici
    [poolLocal drain];
}
```

Nomenclature des accesseurs et mutateurs

```
-(NSString*) maChaine;
-(void) setMaChaine:(NSString*)nouvelleChaine;
```

Les *accesseurs* (*getters*) et les *mutateurs* (*setters*) permettent d'accéder et de modifier les variables d'instance d'une classe sans avoir à les rendre publiques. Cette technique basique de la programmation orientée objet permet d'améliorer l'encapsulation des données et devrait être familière de tout développeur ayant déjà codé en C++, C# et Java.

Maintenant que nous avons vu le fonctionnement de la gestion de la mémoire, il est important d'apprendre à écrire correctement les couples *getter/setter* car c'est là qu'en général les problèmes de fuite de mémoire ou de plantage trouvent leur source.

Supposons que notre classe possède une variable d'instance `maChaine`, de type `NSString`. Alors, la nomenclature des *getter/setter* est la suivante :

- L'accesseur porte le nom de la variable qu'elle représente et commence par une minuscule : `maChaine`.
- Le mutateur porte le nom de la variable avec la première lettre en majuscule, préfixée par `set`: `setMaChaine`.

Pour les getters et les setters simples, Apple recommande trois différentes techniques que nous allons voir dans les sections suivantes. Chacune de ces techniques est appropriée à une situation donnée.

Veuillez noter qu'avec Objective-C 2.0, des directives compilateurs permettent la génération automatique de "propriétés" qui sont une version améliorée des getters/setters (les propriétés devraient être familières aux développeurs C# et Python). De plus, en mode de gestion automatique de la mémoire, les getters/setters ne s'écrivent pas de la même manière. Nous allons voir ces différences au cours de la section suivante.

Écrire des accesseurs et mutateurs

Nous allons décrire ci-dessous les quatre techniques qui permettent d'écrire correctement les accesseurs et mutateurs. Nous vous donnons des indications sur les cas d'utilisations typiques, et nous espérons qu'elles vous seront utiles lorsque vous aurez à choisir la technique correspondant à chaque implémentation.

Technique 1

La première technique consiste à avoir un accesseur qui retient la valeur, puis lui envoie le message `autorelease`, tandis que le mutateur libère la précédente valeur et retient ou copie la nouvelle.

L'accesseur retient, puis `autorelease` l'objet afin de le passer à l'objet d'où provient l'appel. Un `autorelease` seul ne suffit pas, car sinon la variable d'instance serait libérée. De plus, un `release` ne convient pas non plus, puisque la libération serait immédiate et l'appelant se retrouvait avec un pointeur nul (recette pour avoir un plantage immédiat assuré). La seule solution est donc `retain`, puis `autorelease` :

```
- (NSString *) maChaine {
    return [[_maChaine retain] autorelease];
}
```

Le mutateur de son côté est bien plus simple : si le nouvel objet est différent, l'ancien est libéré et le nouveau est retenu.

```
- (void) setMaChaine: (NSString*)
➥nouvelleChaine {
    if (_maChaine != nouvelleChaine) {
        [_maChaine release];
        //il faut soit retenir soit copier le
        //nouvel objet, suivant le besoin
        _maChaine = [nouvelleChaine retain];
    }
}
```

Cette première technique dispose d'un setter rapide et d'un getter relativement lent. Elle est donc à privilégier lorsque le setter est beaucoup plus fréquemment utilisé que le getter.

Technique 2

La seconde technique est l'exacte symétrique de la première, avec un accesseur qui se contente de renvoyer l'objet et un mutateur qui utilise un `autorelease`, puis un `release`. Ici, l'ancien objet reçoit le message `autorelease` dans le mutateur car d'autres objets peuvent être en train de l'utiliser et qu'une libération immédiate causerait des plantages si les autres objets ne l'avaient pas préalablement retenu :

```
- (NSString*) maChaine {
    return _maChaine;
}
- (void) setMaChaine: (NSString*)
➥nouvelleChaine {
    [_maChaine autorelease];
    _maChaine = [nouvelleChaine retain];
}
```

Cette deuxième technique étant la symétrique de la première, le getter est très rapide, tandis que le setter est beaucoup plus lent. Elle est donc à privilégier lorsque c'est le getter qui est utilisé beaucoup plus souvent que le setter.

Technique 3

Cette technique utilise le mutateur de la première technique et l'accesseur de la seconde, ce qui la rend plus avantageuse en termes de performance.

Toutefois, elle demeure aussi extrêmement dangereuse étant donné le risque de libérer l'objet par surprise, l'utilisateur de votre classe peut alors subir un crash si les limitations de votre couple getter/setter ne sont pas explicitées très clairement. Elle reste toutefois recommandée pour les collections/containers d'objets où la performance est cruciale. Vous savez maintenant pourquoi les containers d'objets ne retiennent pas les instances qu'ils contiennent (voir section "Gestion de l'appropriation par les collections d'objets").

```
- (NSString*) maChaine {
    return _maChaine;
}
- (void) setMaChaine: (NSString*)
➥ nouvelleChaine {
    if (_maChaine != nouvelleChaine) {
        [_maChaine release];
        //il faut soit retenir soit copier
        //le nouvel objet, suivant le
        //besoin (voir section suivante)
        _maChaine = [nouvelleChaine retain];
    }
}
```

Technique 4

La création d'objet immuable comme technique d'encapsulation des données et de protection de l'état interne de l'objet est très fréquente. En effet, il est souvent intéressant de créer des objets immuables lorsqu'ils sont de type valeur (*value* en anglais).

La raison est simple : il est plus simple et moins "dangereux" d'utiliser des objets dont la valeur ne peut pas changer. De plus, le code se retrouve également simplifié car les vérifications à faire sont moins nombreuses. Ainsi, par exemple, les instances de la classe `NSString` sont immuables. Pour avoir des chaînes modifiables, il faut instancier la classe `NSMutableString`. Cette distinction entre les deux types existe également pour les autres types de containers (à ce titre, vous pouvez considérer une chaîne de caractères comme un container de caractères sous forme de liste) : `NSFoundation` dispose de `NSArray/NSMutableArray`, de `NSSet/NSMutableSet`, de `NSDictionary/NSMutableDictionary`, ainsi que d'autres classes moins fréquemment utilisées.

La dernière technique que nous allons voir consiste à écrire l'accesseur et/ou le mutateur à l'aide d'une copie afin de rendre les deux objets indépendants l'un de l'autre : chaque objet peut ainsi modifier les valeurs de ses membres sans

modifier les membres de l'autre objet. C'est donc une technique d'encapsulation très importante.

```
//la méthode retourne une copie de l'objet,
//au lieu de retourner l'objet lui-même.
- (NSString*) maChaine {
    return [[_maChaine copy] autorelease];
}
//la méthode copie l'objet qu'elle reçoit, au
// lieu de la retenir simplement.
- (void) setMaChaine: (NSString*)
➥ nouvelleChaine {
    [_maChaine autorelease];
    _maChaine = [nouvelleChaine copy];
}
```

Info

Une fois que vous avez découvert les getters/setters avec copie, vous vous doutez qu'il est bien sûr possible de réécrire chacune des techniques précédentes en utilisant une copie ou une copie muable. Voici donc l'ensemble des nouvelles possibilités qui s'offrent à vous :

- `release/copy` et `release/MutableCopy`
- `autorealese/copy` et `autorelease/MutableCopy`
- `copy/release` et `copy/autorelease`

Le protocole NSCopying

```
//déclaré dans le fichier NSObject.h
@protocol NSCopying
- (id)copyWithZone:(NSZone *)zone;
@end
```

`NSCopying` est un protocole très simple : il ne déclare qu'une seule méthode, `copyWithZone:` qui a pour objectif de fournir une copie fonctionnelle de l'objet dans la zone mémoire passée en paramètre.

Il y a toutefois quelques règles à connaître :

- `NSObject` implémente la méthode `copy`, et donc tous les objets dérivants de `NSObject` disposent également de cette méthode. L'implémentation de `copy` invoque alors en général `copyWithZone:` et passe la zone par défaut comme paramètre.
- Une copie créée *via* `copyWithZone:` est implicitement retenue par l'émetteur, qui est donc également responsable de sa libération.

Enfin, il existe deux sortes de copies en programmation : les copies superficielles (*shallow copy*) et les copies complètes (*deep copy*).

- Une copie est dite superficielle si les pointeurs des membres de l'objet sont copiés : le pointeur est copié, il n'existe qu'un seul objet représentant la valeur de l'objet pointé.

- Une copie est dite complète si les objets référencés par pointeurs sont eux aussi copiés récursivement : les données représentés par l'objet pointé sont copiées, il existe plusieurs instances de la même classe contenant les mêmes données, à différentes adresses de la mémoire.

Deux cas se présentent alors, lorsqu'il faut implémenter la méthode `copyWithZone:` du protocole `NSCopying` :

- La super-classe n'implémente pas le protocole `NSCopying` et donc la classe n'hérite pas la méthode `copyWithZone:`. Il faut alors implémenter `NSCopying` à la main, avec `alloc` et `init`.
- La super-classe implémente le protocole `NSCopying` et la classe hérite donc de la méthode `copyWithZone:` qu'il faut surcharger afin de fournir une implémentation correcte et complète.

Info

Il existe des subtilités à considérer lors de l'implémentation de `NSCopying` que nous n'aborderons pas ici car le format de l'ouvrage ne s'y prête pas. Toutefois, il est important de bien comprendre ses subtilités afin d'implémenter la méthode `copyWithZone:` de manière optimale. Nous vous suggérons donc de vous reporter à la documentation d'Apple pour de plus amples informations.

Implémenter le protocole NSCopying

Nous allons voir ci-dessous les deux principales techniques à utiliser lorsque vous souhaitez implémenter le protocole NSCopying.

Technique 1

```
- (id) copyWithZone:(NSZone* zone) {
    ClasseGuideDeSurvie * copie =
    ➥[[ClasseGuideDeSurvie allocWithZone:zone]
    ➥init];
    [copie setMaChaine:[self maChaine]];
    return copie
}
```

Dans l'exemple précédent, la classe n'hérite pas NSCopying, il faut utiliser alloc, init, puis une fois la nouvelle instance créée, employer les mutateurs pour affecter les valeurs en provenance de l'objet source vers la copie.

C'est une tâche simple, à condition ne pas oublier de bien affecter toutes les variables d'instance du nouvel objet qui sera la copie à retourner.

Technique 2

Lorsque la classe hérite de `copyWithZone:` car sa super-classe implémente le protocole `NSCopying`, deux cas se présentent.

- **La classe dérivée ne déclare pas de nouvelle variable d'instance.** C'est le cas trivial. Vous n'avez rien à faire car vous héritez automatiquement de la méthode `copyWithZone:` de la classe parente, et la copie qu'elle génère est parfaitement valide.
- **La classe dérivée déclare au moins une nouvelle variable d'instance.** Vous devez alors fournir une nouvelle implémentation de `copyWithZone:` qui appellera la méthode de la super-classe et initialisera la (les) nouvelle(s) variable(s) d'instance.

Dans ce second cas, Apple conseille d'utiliser la méthode `alloc/init` pour les nouveaux membres autant que possible. Mais si vous étiez amené à utiliser `copyWithZone:`, voici ce que écririez :

```
- (id) copyWithZone:(NSZone* zone) {
    ClasseDerivee * copie = (ClasseDerivee *)
    ➥ [super copyWithZone:zone];
    copie->maNouvelleVariable = nil;
    [copie setMaNouvelleVariable:[self
    ➥ maNouvelleVariable]];
    return copie
}
```

Attention

Il est en règle générale assez dangereux de surcharger la fonction `copyWithZone:` lorsque vous n'avez pas accès au code de la super-classe. En effet, la manière dont `copyWithZone:` a été implémentée par la super-classe, va fortement impacter la manière dont vous devez surcharger votre propre implémentation.

C'est là qu'apparaît la fonction (que certains qualifient de démoniaque) `NSCopyObject()`. En effet, cette fonction s'occupe simplement de créer une copie superficielle de l'objet passé en paramètre. Ce qui peut sembler tout à fait inoffensif et sans importance va impacter de manière dramatique votre code : vous êtes forcé d'adapter votre implémentation à celle de la super-classe sans quoi votre programme plantera !

Il s'agit d'un concept avancé et nous vous renvoyons à la documentation d'Apple pour bien comprendre les subtilités qui rentrent en jeu.

Voici la conclusion à retenir pour l'instant : faites très attention lorsque vous dérivez une classe dont vous n'avez pas le code et que vous devez surcharger `copyWithZone:`, l'un des exemples notables étant la classe `NSCell` qui est souvent employée comme classe de base en Cocoa et qui utilise `NSCopyObject()`. C'est d'ailleurs la raison pour laquelle les documentations d'Apple donnent toujours les exemples de `NSCell` lorsqu'ils mentionnent `NSCopyObject()` et `copyWithZone:` !

3

Le passage d'Objective-C 1.0 à 2.0

Mac OS X 10.5, sorti officiellement en octobre 2007, apportait pour la première fois depuis l'histoire de Mac OS X, des changements importants au langage Objective-C. Bien qu'il n'y ait jamais eu précédemment de *version* d'Objective-C, Apple a désormais décidé de marquer la différence avec un numéro de version, et c'est ainsi que naquit Objective-C 2.0 (la version précédente devenant Objective-C 1.0).

Les évolutions apportées sont conséquentes et empruntent fortement, comme nous le verrons, aux avancées apportées par Java et C# : mode de gestion

de mémoire automatique, propriétés, énumérations rapides seront détaillés aux cours des sections suivantes.

Attention

Ce chapitre et le précédent sont interdépendants : il est nécessaire de bien comprendre la gestion de la mémoire pour comprendre certaines notions de ce chapitre – par exemple, comment choisir les propriétés automatiques – de même qu'il est important de connaître certaines notions expliquées ici afin de bien appréhender le chapitre sur la gestion de la mémoire. Une lecture linéaire n'est donc pas conseillée aux développeurs débutants en Objective-C 1.0 ou 2.0.

L'environnement d'exécution moderne

Avec Objective-C 2.0, Apple a introduit le concept d'environnement d'exécution moderne (*modern runtime*) et d'environnement d'exécution historique (*legacy runtime*).

Les différences sont très subtiles, mais impactent toutefois la programmation des propriétés, qui sont la nouvelle manière de déclarer et d'implémenter les getters/setters.

Ce qu'il faut retenir c'est que l'iPhone et les programmes 64 bits sur Mac OS X 10.5 et supérieurs disposent de la version moderne de l'environnement

d'exécution et que tous les autres disposent de la version historique (c'est-à-dire tous les programmes 32 bits, y compris sur Mac OS X 10.5). Par exemple, un programme 32 bits sur Mac OS X 10.6 utilisera le *runtime* historique.

Gestion automatique de la mémoire

Bien connu des développeurs Java et .Net, le *ramasse-miettes* (*garbage collector*) s'occupe de libérer la mémoire automatiquement lorsqu'un objet n'est plus utile. De plus, bien qu'il n'y ait pas de ramasse-miettes défini dans le standard C++, des tierces-parties en ont développé et il n'est pas impossible que les développeurs C++ soient familiers avec les concepts de la gestion automatique de la mémoire.

La gestion de la mémoire avec Objective-C nécessitant beaucoup d'attention (comme avec tout langage où la gestion est manuelle) et étant la principale cause d'instabilité des applications Cocoa, le mode ramasse-miettes fut très chaleureusement accueilli par les développeurs Objective-C. Notez bien que l'instabilité provient des erreurs de programmation, et non, bien sûr, de l'environnement d'exécution d'Objective-C.

De plus, Apple n'ayant pas l'habitude de faire les choses à moitié, et disposant de l'expérience de Java et .Net, des méthodes évoluées ont été publiées et un ramasse-miettes à la pointe de la technologie a été créé.

Enfin, et à la différence de Java et .Net, où le ramasse-miettes fait partie intégrante du langage et de l'environnement d'exécution, Apple a créé le premier langage où le mode de gestion automatique de la mémoire est non seulement facultatif (le langage peut fonctionner en gestion manuelle ou automatique de la mémoire). Mais surtout, il possible pour une application ou une bibliothèque de choisir au moment de l'exécution le mode dans lequel tourner. C'est d'ailleurs le seul langage qui, à notre connaissance, permet de faire tourner du code compilé dans les deux modes, de manière tout à fait transparente.

Info

Si vous découvrez Objective-C, ne vous dites pas que vous allez utiliser le ramasse-miettes et que vous n'aurez jamais à apprendre la gestion manuelle de la mémoire, cela serait une grosse erreur. En effet, le ramasse-miettes n'est disponible qu'à partir de Mac OS X 10.5 et vos applications ne fonctionneraient que sur une partie de la base installée des Macintosh : les utilisateurs de Mac OS X 10.5 (Leopard) et 10.6 (Snow Leopard).

Mais surtout, vous vous priveriez d'une base de 40 à 50 millions d'utilisateurs : les adaptes d'iPhone et d'iPod Touch ! En effet, iPhone OS, même dans la version 3.0, ne propose toujours pas le mode de gestion automatique de la mémoire. Apple ayant déjà vendu plus de 30 millions d'iPhone (et plus de 20 millions d'iPod Touch), vous comprenez qu'il est encore très important de savoir gérer la mémoire manuellement.

Pour aller plus loin

Le ramasse-miettes utilisé par l'environnement d'exécution d'Objective-C 2.0 s'appelle *AutoZone* et n'est en fait pas limité à Objective-C. Il peut être implémenté pour n'importe quel langage et est par exemple également utilisé par l'implémentation de MacRuby.

Apple en a fait un *logiciel libre* sous la licence Apache 2.0. Cela signifie que non seulement son code source est disponible, mais que tout développeur ou toute entreprise peut utiliser le code gratuitement dans ses propres produits en respectant les termes de la licence.

Vous pouvez télécharger le code depuis http://www.opensource.apple.com/source/autozone/.

Avantages et inconvénients du ramasse-miettes

L'avantage le plus évident est que la gestion de la mémoire est grandement simplifiée, et que la plupart des problèmes liés à une mauvaise gestion de la mémoire sont résolus sans le moindre effort de la part du développeur.

L'un des problèmes de gestion de mémoire auquel il fallait être particulièrement attentif, les cycles de rétention, est facilement résolu puisque le ramasse-miettes détecte les objets inutiles, même s'ils forment un cycle de rétention. En effet, ces objets ne sont pas accessibles depuis les objets racines.

Info

Nous n'avons pas évoqué le sujet des accesseurs et des mutateurs lorsque le code doit être *thread-safe*[1] car les développeurs experts pourront résoudre le problème une fois qu'ils ont les bases pour écrire les accesseurs et mutateurs en mode mono-thread. Toutefois, ils nécessitent la plus grande attention et l'utilisation de verrouillage (*locking*) afin d'assurer leur atomicité. Ils ont donc un impact négatif sur la performance du code.

Ceci est donc un problème à moitié résolu puisque les accesseurs n'ont plus besoin de verrouillage et nous avons même droit à un gain en termes de performance.

Les inconvénients sont relativement minimes comparés aux avantages :

- Il est possible d'avoir un impact négatif en termes de performance lors de l'allocation d'un grand nombre d'objet et lors du passage du ramasse-miettes, mais les tests sous Java et .Net ont montré qu'en général, l'impact est négligeable.
- L'application requiert davantage de mémoire lorsqu'il fonctionne. Ceci est dû au fait que la mémoire n'est pas libérée immédiatement et devrait être quasiment négligeable.
- Il faut modifier la manière de penser le code, implémenter la méthode `finalize` (et `invalidate` quand nécessaire).

1. Le terme anglais *thread-safe* signifie que le code peut être exécuté simultanément depuis plusieurs threads sans remettre en cause l'intégrité des objets en mémoire ni compromettre l'état de l'application.

Vous remarquerez que les avantages semblent bien plus nombreux que les inconvénients. Cette conclusion n'en est pas moins tout à fait logique : il est difficile d'imaginer qu'Apple aurait non seulement perdu tant de temps à implémenter un ramasse-miettes, mais qu'en plus ils auraient décidé de l'intégrer au langage si les avantages n'étaient pas conséquents !

Fonctionnement du ramasse-miettes

Le ramasse-miettes d'Objective-C fonctionne exactement sur le même principe que celui de Java et .Net. La fonction principale d'un ramasse-miettes est de trouver les objets qui ne sont plus utilisés afin de libérer la mémoire qu'ils occupent. Or, s'il venait à libérer un objet qui devait être utilisé par la suite, cela résulterait dans le plantage immédiat de votre application. Le ramasse-miettes n'a donc pas le droit à l'erreur : il doit être sûr et certain qu'un objet ne sera plus utilisé avant de libérer l'espace qu'il occupe en mémoire.

Comment fait-il donc pour savoir quels objets il peut libérer ? En pratique, le ramasse-miettes s'occupe d'abord de repérer l'ensemble des objets utilisés, puis libère les autres. Il emploie à cette fin la même technique que le ramasse-miettes de .Net et Java : il commence par scruter les objets faisant partie de

la *racine* de l'application : les variables globales et les variables de la pile, ainsi que les objets ayant des références externes à l'application et ceux utilisés par les différents *threads*.

En parcourant les objets pointés (par des *références fortes*) par ces *objets racines*, il établit un graphe de tous les objets atteignables (*reachable*). Tous les objets non-référencés sont alors considérés comme inutiles. Ils sont mis de côté et seront alors finalisés – c'est-à-dire que leur méthode `finalize` sera appelée – avant de voir leur mémoire libérée automatiquement à la fin du processus.

Référence forte et référence faible

Il existe en Objective-C – de même qu'en Java et C# – deux sortes deux références : les *références fortes* (*strong reference*) et les *références faibles* (*weak reference*). Les références sont fortes par défaut.

Comme nous l'avons vu ci-dessus, une référence forte implique que le ramasse-miettes va ajouter l'objet à la liste des objets utilisés et ne tentera pas de le libérer.

En revanche, un objet lié à un autre par une référence faible sera considéré par le ramasse-miettes comme un objet libérable. La référence faible est une technique d'optimisation de l'utilisation de la mémoire et n'est pas très utilisée (si ce n'est dans les containers prévus pour, tel que `NSMapTable`, `NSMapTable` et `NSPointerArray`).

Son utilisation est simple : si vous savez qu'un objet occupe beaucoup de place en mémoire, qu'il n'est pas utilisé fréquemment et qu'il est facile de le reconstituer, vous emploierez une référence faible vers cet objet avec le mot-clé `__weak`. Ainsi, si le ramasse-miettes passe, il libérera la mémoire occupée. Comme vous ne savez pas quand le ramasse-miettes effectue son travail, vous devez prévoir un test de nullité sur l'objet avant toute utilisation pour savoir si l'objet a été libéré ou pas, et le régénérer au cas où.

Un autre exemple important, donné par Apple, est le cas de `NSNotificationCenter` : les centres de notifications recourent aux références faibles vers leurs abonnés. En effet, un abonné non utilisé peut alors être libéré. Si la référence n'était pas faible, la vie de l'objet serait liée à celle du centre de notifications et il ne serait jamais libéré.

Info

Vous avez sans doute relevé que les objets sont finalisés avant d'être libéré automatiquement. La méthode `finalize` en Objective-C (et C# et Java) est l'équivalent du destructeur du C++ et de la méthode `dealloc` en mode manuel.

Nous reviendrons sur cette méthode à la section "Implémenter la méthode finalize" qui lui est dédiée.

Nouveaux paradigmes à adopter avec le ramasse-miettes

Apple recommande de n'utiliser le ramasse-miettes que pour les nouveaux projets. En effet, mettre à jour un code existant pour le ramasse-miettes demande beaucoup d'effort (pensez à tous les `retain`, `release`, `dealloc`, `autorelease` qu'il va falloir trouver et changer !) et s'avère donc être une opération assez périlleuse (le moindre oubli sera puni par des fuites mémoires ou des plantages).

Il n'y a donc pas beaucoup d'intérêt à vouloir migrer une application qui fonctionne parfaitement bien et dont vous maîtrisez le code.

L'utilisation du ramasse-miettes simplifie beaucoup la gestion de la mémoire, mais elle implique également la compréhension et l'utilisation de quelques nouveaux paradigmes que nous allons évoquer ci-dessous.

La méthode `finalize`

```
- invalidate {
    if (monFichier != nil) {
                    //monFichier est une instance
                    //de NSFileHandle
        [monFichier closeFile]
    }
}
- finalize {
    [self invalidate]
    [super finalize]
}
```

Les langages disposant d'un ramasse-miettes (Java, C# et Objective-C) utilisent la méthode `finalize` au lieu de destructeur (comme C++).

Cette méthode est appelée par le ramasse-miettes juste avant que la mémoire allouée à l'objet ne soit récupérée. Elle permet donc de libérer les dernières ressources et ce à la dernière minute.

La section "Implémenter la méthode `finalize`" est dédiée à cette méthode et nous vous invitons à vous y reporter.

Libérer les ressources chères dès que possible

```
- invalidate{
    if (monFichier != nil) {
                     //monFichier est une instance
                     //de NSFileHandle
        [monFichier closeFile]
    }
}
```

Une distinction importante existe entre la gestion manuelle de la mémoire et la gestion automatique : lorsque le ramasse-miettes travaille à votre place, son action est non déterministe et vous ne savez jamais quand surviendra son prochain passage.

Maintenant, imaginez que vous avez écrit une application qui tourne sur un serveur de manière continue et que cette application instancie une classe qui contient des pointeurs vers de nombreux

fichiers (comme par exemple un serveur de fichiers où chaque connexion serait ainsi gérée). Il se peut alors que le ramasse-miettes ne passe pas pendant des jours si votre service n'est que peu sollicité ou bien si la machine dispose d'une très grande mémoire.

Vous devez de libérer les descripteurs de fichier sans attendre que votre objet ne soit ramassé automatiquement. Cela signifie également qu'il ne faut pas attendre que la méthode `finalize` soit appelée.

Java et C# ayant bien sûr rencontré le même problème, la solution qu'ils proposent peut également s'appliquer ici. La classe implémente une méthode appelée `dispose()` qui s'occupe de libérer les ressources chères et d'invalider l'objet. C# dispose même d'une `interface` (l'équivalent du `protocol` d'Objective-C) appelée `IDisposable` et d'un pattern (*IDisposable pattern*) à cette fin.

Apple n'a pas encore proposé de solution standard, mais il semble logique d'utiliser une méthode appelée `dispose` ou même mieux, `invalidate`, pour suivre la nomenclature de la documentation officielle.

Objets obtenus depuis les fichiers nib

Il existe une différence de fonctionnement de l'environnement d'exécution entre le mode géré et le mode automatique qui impacte les objets obtenus depuis les fichiers nib (*nib files*) d'Interface Builder. Assurez-vous de bien avoir des références fortes vers ces objets (tels que par exemple les objets

contrôleurs). En effet, sans référence forte, ces objets vont être nettoyés par le ramasse-miettes.

Même si en pratique ce problème doit être extrêmement rare, Apple recommande de les connecter *via* l'attribut File's Owner.

Activer le ramasse-miettes

L'activation du ramasse-miettes se fait au moment de la compilation, et c'est donc une option à passer au compilateur – sous forme de drapeau dans le cas de GCC. Il est toutefois recommandé de faire ce réglage dans Xcode si vous utilisez l'environnement de développement gratuit d'Apple (voir Figure 3.1).

Il existe trois modes différents pour la gestion automatique de la mémoire :

- *Unsupported* (*non compatible* ; pas de drapeau passé au compilateur). Le code n'est pas écrit pour la gestion automatique de mémoire et est donc incompatible avec le ramasse-miettes.
- *Required* (*nécessaire* ; le drapeau `-fobjc-gc-only` est passé au compilateur). Le ramasse-miettes est requis car le code ne gère pas la mémoire autrement (`retain`/`release`). Si c'est une bibliothèque, cela signifie qu'elle ne pourra pas être utilisée par une application qui tourne en mode de gestion manuelle. S'il s'agit d'une application, cela implique qu'elle ne s'exécutera que sur Mac OS X 10.5 et supérieurs (voir note si après).

- *Supported* (*compatible* ; le drapeau `-fobjc-gc` est passé au compilateur). Cela signifie que le code peut s'exécuter en mode ramasse-miettes, mais également en mode géré. Les méthodes `finalize`, ainsi que `retain` et `release`, sont alors utilisées, ce qui rend l'application, ou la bibliothèque, réellement *universelle*.

Figure 3.1

Pour aller plus loin

Objective-C 2.0 n'étant disponible que sur Mac OS X 10.5 et supérieur, tout code écrit en Objective-C 2.0 ne fonctionne en théorie que sur 10.5 et supérieur. Utiliser ou non le ramasse-miettes ne sera pas un choix très difficile : soit vous codez `retain/release` et gérez la mémoire manuellement soit vous utilisez le mode GC.

Le choix est trivial sur iPhone OS où vous n'avez pas de ramasse-miettes (pour l'instant...).

De manière générale, Apple recommande de réserver le mode mixte (*dual-mode*) pour les bibliothèques et les *frameworks* car vous ne savez dans quel mode vos clients fonctionnent.

Les développeurs avancés seront heureux d'apprendre qu'il est possible d'écrire du code qui fonctionnera en mode GC sous 10.5 et en mode manuel sous 10.4. Il faut bien sûr se passer des références faibles (`__weak`) ainsi que des containers les utilisant (`NSMapTable`, `NSHashTable` et `NSPointerArray`).

Andre Pang a écrit un long article sur ce sujet trop avancé pour être traité dans cet ouvrage. Nous vous en recommandons la lecture sur MDN : http://www.mac-developer-network.com/podcasts/lnc/lnc036/.

Détecter que le code s'exécute en mode GC

```
if ([NSGarbageCollector defaultCollector] != nil)
{
        // exécution en mode GC
}
else {
        // mode manuel : il faut utiliser
        // retain/release, etc.
}
```

Vous déterminez si le code s'exécute ou non en mode GC avec le code ci-dessus.

Si vous êtes en train d'écrire du code destiné à être exécuté en mode GC et en mode manuel ou si vous développez une bibliothèque, vous devez écrire deux fois le code relatif à la gestion de la mémoire, et vous devez exécuter le code approprié lors de l'exécution.

Le code précédent ne fonctionne bien sûr qu'à partir de Mac OS X 10.5 puisque l'objet-classe `NSGarbageCollector` n'existe pas dans les versions précédentes.

Pour aller plus loin

Vous pouvez sans doute obtenir le même résultat grâce avec un appel à `objc_getClass("NSGarbageCollector")` qui fonctionne indépendamment de la disponibilité de

la classe `NSGarbageCollector`. Il est toutefois donné ici à être éducatif seulement, car il est fortement déconseillé d'utiliser ce genre d'astuce pour obtenir une rétrocompatibilité.

Solliciter le ramasse-miettes

```
NSGarbageCollector *collector =
➥ [NSGarbageCollector defaultCollector];
[collector collectIfNeeded];
                        // nettoie la mémoire si
                        // bonne opportunité
//ou :
[collector collectExhaustively];
                        // force un nettoyage
                        // exhaustif de la mémoire
```

Ce code montre qu'Objective-C dispose d'un moyen d'indiquer au ramasse-miettes qu'un petit nettoyage serait bienvenu.

Java et C# disposent également de cette possibilité de suggérer un nettoyage, voire de forcer le ramasse-miettes de procéder à un nettoyage de la mémoire, avec respectivement `System.gc()` et `GC.Collect()`. Mais leur utilisation est fortement déconseillée, sauf cas extrêmement spécifiques.

Attention

De la même manière qu'en Java et C# donc, il est déconseillé d'utiliser cette méthode en Objective-C sauf dans des cas très spécifiques et très rares. En effet, il arrive qu'en voulant ainsi optimiser une application, le développeur obtienne l'effet inverse.

Le ramasse-miettes est actionné par l'environnement d'exécution lorsque ce dernier trouve une bonne opportunité pour le faire. Cette opportunité dépend non seulement de l'état de l'application en cours, mais également de données externes auxquelles le développeur n'a pas accès (telles que la quantité de mémoire disponible sur le système, la mémoire occupée par les autres applications, l'utilisation du processeur, etc.)

Un bon ramasse-miettes va optimiser les ramassages en fonction de toutes ces données et utiliser une heuristique à cette fin.

Implémenter la méthode `finalize`

```
- invalidate {
    if (monFichier != nil) {
                //monFichier est une instance
                //de NSFileHandle
        [monFichier closeFile]
    }
```

```
}
- finalize {
    [self invalidate]
    [super finalize]
}
```

La méthode `finalize` est la méthode jumelle de `dealloc` lorsque la gestion automatique de la mémoire est activée.

La règle générale est que si vous avez besoin d'une méthode `finalize`, c'est qu'il vous faut également une méthode `invalidate` qui sera appelée pour libérer les ressources de votre instance avant la finalisation.

Comme vous le voyez, il n'y a rien de difficile dans l'écriture de la méthode `finalize` puisque tout le travail doit être fait dans la méthode `invalidate`.

Attention

Il est important de bien comprendre les risques inhérents à la mauvaise utilisation de la méthode `finalize`.

Tout comme le ramasse-miettes de la machine virtuelle Java ou de l'environnement d'exécution commun de .Net, la finalisation des objets se fait dans un ordre non déterminé. Il est donc extrêmement important d'éviter autant que possible tout contact avec d'autres objets *finalisables* dans l'implémentation de la méthode `finalize` afin d'éviter les plantages, voire les résurrections, d'objets déjà finalisés.

De plus, étant donné que tous les objets ramassés voient leur méthode **finalize** appelée lors du processus, il est important que chaque méthode exécute le moins de tâches et soit la plus courte possible, afin de minimiser le blocage de l'application durant le ramassage.

Idéalement, la méthode **finalize** ne devrait rien faire et s'il y a des ressources à libérer, il faut le faire dans la méthode **invalidate** (voir la section "Libérer les ressources chères dès que possible").

Écrire les accesseurs et mutateurs en mode GC

```
//couple getter/setter sans copie
- (NSString*) maChaine {
    return _maChaine;
}
- (void) setMaChaine: (NSString*)
➥ nouvelleChaine {
    _maChaine = nouvelleChaine
}
//couple getter/setter avec copie
//la méthode retourne une copie de l'objet,
//au lieu de retourner l'objet lui-même.
- (NSString*) maChaine {
    return [_maChaine copy];
}
//la méthode copie l'objet qu'elle reçoit,
//au lieu de la retenir simplement.
```

```
- (void) setMaChaine: (NSString*) nouvelleChaine {
    if (_maChaine != nouvelleChaine) {
        _maChaine = [nouvelleChaine copy];
}
```

L'activation du ramasse-miettes supprime toutes les difficultés relatives à l'écriture des accesseurs et des mutateurs, à tel point que leur écriture devient triviale. Nous avons vu au chapitre précédent que, suivant le besoin, il y avait trois différentes manières d'écrire les accesseurs et mutateurs simples et également une quatrième technique lorsque vous souhaitez copier la variable d'instance.

En mode ramasse-miettes, tout ceci se simplifie en un couple getter/setter simple et un couple getter/setter pour la copie, sans aucune difficulté.

Info

L'écriture des accesseurs/mutateurs est tellement simple en mode ramasse-miettes qu'il n'est même plus la peine de les écrire. Nous allons voir dans les sections suivantes comment demander au compilateur de les générer pour nous. Les propriétés auto-générées devraient couvrir au moins 95 % besoin. Vous n'aurez à implémenter le couple getter/setter que lorsque vous souhaitez effectuer des opérations spéciales à chaque fois qu'un getter ou un setter est appelé.

Utiliser la nouvelle Dot Syntax

```
@interface ClasseGuideDeSurvie : NSObject {
//… définitions diverses
}
- (void) setMaChaine: (NSString *)chaine;
- (NSString *) maChaine;
@end
// …
ClasseGuideDeSurvie * instance =
➥ [[ClasseGuideDeSurvie alloc] init];
instance.maChaine = @"chaine";
NSString * test = instance.maChaine;
```

Cette nouvelle syntaxe utilisant un point (`.`) au lieu des crochets habituels (`[ ]`), appelée *Dot Syntax*, est très proche des appels de méthodes en Java et C#.

Non seulement, elle améliore la lisibilité du code, notamment lorsque plusieurs messages doivent être imbriqués, mais elle apporte même quelques avantages. Son utilisation est particulièrement appropriée pour les propriétés, que nous découvrons dans la section suivante.

Les appels sous forme de Dot Syntax sont transformés en messages traditionnels par le compilateur. Il n'y a donc techniquement aucune différence de code entre un envoi de message traditionnel et la nouvelle syntaxe. Toutefois, cette nouvelle syntaxe permet d'effectuer des vérifications supplémentaires par rapport à l'envoi de message : c'est notamment le cas lorsque vous essayer de lire une propriété

qui n'existe pas, d'affecter une propriété qui n'existe pas, ou de changer la valeur d'une propriété qui ne possède est en lecture seule (c'est-à-dire qu'il y a un getter, mais pas de setter associé). Dans ces cas précis, le compilateur génère une erreur alors qu'un envoi de message invalide n'engendre qu'un avertissement. Le fonctionnement est très simple :

```
instance.maPropriete = valeur;
id resultat = instance.maPropriete;
```

Par défaut, lorsque le compilateur rencontre une lecture de la valeur `instance.maPropriete`, il appelle la méthode `maPropriete` et lorsqu'il rencontre une affectation, il la transforme en un appel de la méthode `setPropriete:`.

Notez enfin qu'il est possible de changer le comportement par défaut en utilisant les propriétés. Nous découvrons les propriétés à la section suivante.

Attention

La Dot Syntax réclame au développeur de la discipline. Étant donné que la Dot Syntax n'est qu'une sorte d'astuce du compilateur, il est possible d'en user et d'en abuser. C'est pourquoi, il est recommandé au développeur de rester discipliné et de limiter la syntaxe aux getter/setter. En effet, rien n'empêche d'utiliser la syntaxe pour n'importe quelle méthode et de voir des lignes de code horribles, telles que `instance.retain` qui est valide, mais contraire à l'esprit de la syntaxe.

Déclarer les propriétés

```
// la propriété suivante est totalement
// équivalente aux getters/setters qui sont
// commentés ci-dessous :
@property NSString * maChaine;
//- (void) setMaChaine: (NSString *)chaine;
//- (NSString *) maChaine;
```

Les propriétés sont formées de deux parties : une partie *déclaration* et une partie l'*implémentation*. Le code ci-dessus montre ce à quoi ressemble la partie déclaration. Nous traiterons l'implémentation dans la section suivante.

Les propriétés sont l'une des nouveautés importantes d'Objective-C 2.0 et permettent, non seulement de simplifier grandement la tâche lors de l'écriture des accesseurs/mutateurs, mais aussi de spécifier clairement aux utilisateurs de vos classes les choix ayant été fait pour l'implémentation de vos méthodes.

Info

Les propriétés sont disponibles en C# depuis la version 1.0, mais elles ont été affinées avec la version 2.0, puis 3.0 du langage. Python dispose également des propriétés qui sont déclarées grâce à des décorateurs. En revanche, Java n'a jamais adopté les propriétés, ce qui est assez dommage car les développeurs Java ne peuvent pas profiter de leurs avantages.

Les propriétés sont tellement importantes qu'Apple préconise désormais de ne pas déclarer les accesseurs et mutateurs autrement que par les propriétés.

Une fois qu'une propriété a été ainsi déclarée, vous avez le choix entre fournir vous-même l'implémentation ou bien demander au compilateur de la générer pour vous. Dans la très grande majorité des cas, il est totalement inutile de passer votre temps à les écrire et vous demanderez donc au compilateur de travailler à votre place et vous pouvez spécifier, grâce aux attributs de la propriété, les règles qu'il devra suivre afin d'implémenter les méthodes suivant vos besoins (nous consacrons la section suivante à ces attributs).

Les attributs des propriétés

```
@property (attributs) type nom;
```

Par défaut, les propriétés créées porteront respectivement le nom de `nom/setNom` pour le getter et le setter. Il est possible de proposer au compilateur d'autres noms, comme nous le verrons dans la section suivante, sous forme d'attributs. De plus, `attributs` est constitué d'une liste d'attributs séparés par des virgules.

Un exemple concret aurait donc la forme :

```
@property (readwrite, assign, atomic) NSString *
➥ maChaine;
```

L'exemple ci-dessus est tout à fait équivalent au code ci-après :

```
//- (void) setMaChaine: (NSString *)chaine;
//- (NSString *) maChaine;
```

Toutefois, la propriété dispose d'un avantage important : elle permet à l'utilisateur de la déclaration de savoir que votre propriété utilise l'assignation (par opposition à la copie avec `copy` ou à la retenue avec `retain`) et que vos méthodes sont atomiques (c'est-à-dire *thread-safe*).

Spécifier le nom des accesseurs et mutateurs d'une propriété

```
@property (getter=maChaine) NSString * maChaine;
@property (setter=setMaChaine) NSString *
➥ maChaine;
@property (getter=maChaine, setter=setMaChaine)
➥ NSString * maChaine;
```

Vous pouvez spécifier le nom des accesseurs et mutateurs grâce aux attributs `getter=` et `setter=`, qui sont tous deux facultatifs. Par exemple, les trois déclarations ci-dessus sont valides.

Toutefois, et, en règle générale, il vaut mieux suivre la convention consistant à avoir les propriétés `maChaine` et `setMaChaine:` et éviter de compliquer

les choses en fournissant des méthodes utilisant une autre nomenclature.

Définir une propriété en lecture seule

```
@property NSString * maChaine;  //1
@property (readwrite) NSString * maChaine; //2
@property (readonly) NSString * maChaine; //3
```

Par défaut, les propriétés sont en lecture et écriture, mais vous pouvez utiliser l'attribut `readwrite` pour le rendre explicite et l'attribut `readonly` pour annoncer une propriété en lecture seule.

Dans l'exemple précédent, (1) et (2) sont totalement équivalents, tandis que (3) annonce une propriété qui sera en lecture seule.

Info

Comme nous l'avons vu en abordant la Dot Syntax, lorsqu'une propriété est définie en lecture seule, le compilateur génère une erreur si vous essayez de modifier la propriété. Ceci est différent des messages standard où le compilateur se limite à envoyer un avertissement lorsqu'il détecte qu'une méthode est manquante.

Définir le type de comptage de références

```
@property NSString * maChaine; //1
@property (assign) NSString * maChaine; //2
@property (retain) NSString * maChaine; //3
@property (copy) NSString * maChaine; //4
```

Les attributs `assign`, `retain` et `copy` sont disponibles afin de spécifier le fonctionnement interne du mutateur. Par défaut, c'est `assign` qui est choisi.

Dans l'exemple précédent, (1) et (2) sont totalement équivalents et stipulent que le mutateur se limite à assigner la nouvelle valeur, tandis que (3) signifie que le mutateur retient l'objet assigné et la valeur précédente est libéré. Enfin, (4) spécifie que la mutateur obtient une copie de l'objet et libère l'ancienne valeur. Ce dernier attribut requiert que la classe de la propriété implémente le protocole `NSCopying`.

Info

Si vous utilisez le ramasse-miettes, vous ne devez pas spécifier d'attribut ici, sous peine d'avoir une erreur à la compilation. La seule exception est le cas où votre classe implémente le protocole `NSCopying` et que vous utilisez l'attribut `copy`.

Il est recommandé, lorsque la mémoire est gérée manuellement, de ne pas se baser sur la valeur par défaut, mais d'expliciter votre choix en utilisant l'attribut `assign`. Cela rendra votre code d'autant plus clair et vous évitera un avertissement inutile de la part du compilateur.

Spécifier l'atomicité de la méthode

```
@property NSString * maChaine; //1
@property (nonatomic) NSString * maChaine; //2
```

Par défaut, toutes les propriétés sont atomiques. L'attribut `nonatomic` permet de spécifier que le getter et le setter sont non-atomiques.

Dans l'exemple précédent, (1) désigne une propriété atomique (c'est-à-dire que les méthodes sont *thread-safe*). Cela signifie que lorsque la mémoire est gérée manuellement, des verrous (*lock*) doivent être utilisés.

L'exemple (2) désigne une propriété non atomique. Dans ce cas, l'implémentation de l'accesseur se contente de simplement renvoyer la valeur.

Spécifier qu'une référence est forte ou faible pour une propriété

```
@property (attributs) NSString * maChaine; //1
@property (attributs) __strong NSString *
➥ maChaine; //2
@property (attributs) __weak NSString *
➥ maChaine; //3
```

Par défaut, les références sont fortes. Toutefois, vous pouvez utiliser le décorateur `__weak` et `__strong` pour spécifier le type de référence de la propriété.

Dans l'exemple précédent, (1) et (2) sont totalement équivalents et désignent une référence forte, tandis que dans (3) c'est une référence faible qui a été déclarée.

Attention

Comme vous pouvez le voir, `__weak` et `__strong` ne sont pas déclarés avec les attributs. Ce sont des décorateurs et non pas des attributs de la propriété. C'est une différence syntaxique sans vraiment grande importance.

Demander au compilateur d'écrire le code des propriétés

```
// déclaré dans le fichier d'en-tête :
@property (nonatomic) NSString * maChaine;
// fichier d'implémentation :
@implementation ClasseGuideDeSurvie
@synthesize maChaine;
// …
@end
```

Comme vous pouvez le voir, en une seule ligne et grâce à la directive `@synthetize`, le compilateur génère l'implémentation de la méthode automatiquement. Les développeurs .Net reconnaîtront cette fonctionnalité disponible depuis la version 3.0 de C# et portant le nom de *propriété auto-implémentée*.

Cette directive demande au compilateur de générer à la volée le code du getter, et si votre propriété n'est pas marquée par l'attribut `readonly`, le code du setter (`readwrite` est l'attribut par défaut).

Il est aussi important de noter que vous n'êtes pas obligé d'utiliser `@synthetise`. Vous pouvez bien sûr fournir votre propre implémentation de l'accesseur et du mutateur, même si vous avez déclaré les propriétés avec `@property` dans le fichier d'en-tête.

Si vous avez plusieurs couples de getter/setter à implémenter, vous pouvez les mettre à la suite sur une seule ligne, de la manière suivante :

```
@synthesize maChaine, monAutreChaine;
```

De plus, bien que nous ne recommandions pas cette technique, il est possible de spécifier la variable d'instance à utiliser pour la propriété :

```
@synthesize maChaine = uneChaineQuelconque;
```

Le code précédent demande au compilateur de générer les méthodes `maChaine` et `setMaChaine:` en utilisant la variable d'instance `uneChaineQuelconque`. Il est plus propre et plus simple de toujours utiliser une variable et les getter/setter portant le même nom.

Info

La directive `@dynamic` peut être utilisée à la place de `@synthetize`, voir section "Fonctionnement de @dynamic".

Runtime moderne et Runtime historique

Si votre code est compilé sur un environnement d'exécution historique, la variable d'instance sur laquelle se base la propriété doit avoir été déclarée dans le fichier d'en-tête.

En revanche, si votre code est compilé sur un environnement d'exécution moderne, il n'est pas obligatoire de déclarer la variable d'instance : si le compilateur le trouve, il l'utilise, s'il ne le trouve pas, il le génère également en même temps que la propriété.

N'oubliez pas la méthode `dealloc` si vous gérez manuellement la mémoire. Comme le compilateur génère pour vous les getter/setter, il est tentant d'oublier que la tâche d'écrire la méthode `dealloc` vous incombe toujours.

Fonctionnement de @dynamic

```
@implementation ClasseGuideDeSurvie
@dynamic maChaine;
- (NSString*) maChaine {
    return _maChaine;
}
- (void) setMaChaine: (NSString*) nouvelleChaine {
    if (_maChaine != nouvelleChaine) {
        [_maChaine release];
        _maChaine = [nouvelleChaine retain];
    }
}
@end
```

Dans l'exemple précédent, vous pouvez constater que la directive `@dynamic` est utilisée, mais que l'implémentation a été également fournie.

La directive `@dynamic` est assez originale puisqu'elle ne possède vraiment pas d'équivalent dans les autres langages : C# ne dispose pas encore de propriétés dynamiques dans la version 3.0, mais la version 4.0 du langage les intègre (sortie en 2010). Python dispose du concept de propriétés, mais il faut appliquer des annotations, donc la définition doit être présente même s'il est possible de la remplacer pendant l'exécution.

Si vous utilisez `@dynamic` au lieu `@synthetize`, vous faites comprendre au compilateur que vous vous chargez de fournir l'implémentation, et que ce dernier n'a pas à se soucier de trouver l'implémentation dans votre code : l'implémentation peut être fournie au moment de la compilation *ou* dynamiquement, lors de l'exécution par un mécanisme tel que le chargement dynamique de code ou la résolution dynamique de méthode.

C'est bien entendu une méthode avancée qui ne sera pas forcément employée dans les projets relativement simples.

Déclarer une propriété avec un getter public et un setter privé

```
//fichier d'en-tête: "ClasseGuideDeSurvie.h"
@interface ClasseGuideDeSurvie : NSObject
➥ <MonProtocole, NSCopying> {
    @private
    NSString * maChaine;
```

```
}
@property (nonatomic, readonly, assign)
➥ NSString * maChaine;
@end
//Extension privée :
➥ "ClasseGuideDeSurvie+PrivateMethodsExtension.h"
@interface ClasseGuideDeSurvie ( )
@property (readwrite, assign, nonatomic)
➥ NSString * maChaine;
@end
//fichier d'implémentation "ClasseGuideDeSurvie.m"
#import "ClasseGuideDeSurvie.h"
@implementation ClasseGuideDeSurvie
@synthesize maChaine
@end
```

L'une des spécificités importantes des propriétés est qu'il est possible de déclarer une propriété `readonly` dans le fichier d'en-tête d'une classe, puis de la redéclarer comme `readwrite` dans une extension de la classe (voir section "Les extensions" au Chapitre 1, pour savoir ce qu'est une extension).

Il est ainsi possible, comme dans l'exemple précédent, de déclarer une propriété dont le getter est public et le setter privé. Cette technique fort utile devrait être plus simple à implémenter : les développeurs C# 2.0 peuvent faire la même chose de manière bien plus simple grâce au fait que C# n'a pas besoin de fichier d'en-tête.

Créer une sous-classe muable d'une classe immuable

```
@interface ClasseImmuable : NSObject {NSString *
➥ maChaine;}
@property (readonly, copy) NSString * maChaine;
@end
@interface ClasseMuable : ClasseImmuable {}
@property (readwrite, copy) NSString * maChaine;
@end
@implementation ClasseImmuable
@synthesize maChaine;
@end
@implementation ClasseMuable
@synthesize maChaine;
@end
```

La seconde spécificité importante des propriétés et qu'il est possible pour une sous-classe de redéclarer une propriété `readonly` de la classe mère et de la rendre ainsi `readwrite`. Tous les autres attributs doivent rester les mêmes.

Bien que cela puisse paraître étrange, voire choquant à première lecture, cela devient tout à fait logique dès que l'on se rappelle qu'une propriété n'est rien de plus qu'un couple de méthodes (et en l'occurrence, parfois une méthode unique).

Il est donc tout à fait normal pour une sous-classe de pouvoir ajouter une méthode à l'ensemble des méthodes héritées de sa super-classe, en l'occurrence un setter.

Info

Un objet est dit *muable* lorsque ses propriétés peuvent changer. Par exemple un dictionnaire est dit muable s'il est possible d'y insérer des éléments (associés à des nouvelles clés), ou de changer les éléments correspondants aux clés déjà contenues dans le dictionnaire.

À l'opposé, un objet est dit *immuable* lorsqu'aucune de ses propriétés ne peut changer. Si nous reprenons l'exemple du dictionnaire, la version immuable serait un dictionnaire qui, une fois initialisé, serait un mode lecture seule : impossible d'insérer ou de supprimer des objets ou de modifier les objets correspondants aux clés présentes.

Utiliser les énumérations rapides

```
NSDictionary* dic = [NSDictionary
➥ dictionaryWithObjectsAndKeys: @»clé1",
➥ [NSNumber numberWithInt:1], @"clé2",
➥ [NSNumber numberWithInt:2], @"clé3",
➥ [NSNumber numberWithInt:3], nil];
for(id elem in dic){
    NSLog(@"boucle1 - élément: %@ de type %@»,
     ➥elem, [elem className]);
}
for(id elem in [dic allKeys]){
    NSLog(@"boucle2 - élément: %@ de type %@»,
    ➥ elem, [elem className]);
}
for(id elem in [dic allValues]){
    NSLog(@"boucle3 - élément: %@ de type %@»,
    ➥ elem, [elem className]);
```

```
}
for(id elem in [dic keyEnumerator]){
NSLog(@"boucle4 - élément: %@ de type %@»,
➥ elem, [elem className]);
}
for(id elem in [dic objectEnumerator]){
        NSLog(@"boucle5 - élément: %@ de type
        ➥%@», elem, [elem className]);
}
```

L'exécution du code précédent donne :

```
boucle1 - élément: 1 de type NSCFNumber
boucle1 - élément: 2 de type NSCFNumber
boucle1 - élément: 3 de type NSCFNumber
boucle2 - élément: 1 de type NSCFNumber
boucle2 - élément: 2 de type NSCFNumber
boucle2 - élément: 3 de type NSCFNumber
boucle3 - élément: clé1 de type NSCFString
boucle3 - élément: clé2 de type NSCFString
boucle3 - élément: clé3 de type NSCFString
boucle4 - élément: 1 de type NSCFNumber
boucle4 - élément: 2 de type NSCFNumber
boucle4 - élément: 3 de type NSCFNumber
boucle5 - élément: clé1 de type NSCFString
boucle5 - élément: clé2 de type NSCFString
boucle5 - élément: clé3 de type NSCFString
```

Les énumérations rapides sont l'équivalent des boucles `foreach` de C# et leur équivalent sous forme de boucle `for` modifiée en Java et Python.

Vous pouvez utiliser les énumérations rapides avec toutes les classes containers de Cocoa, Apple ayant pris soin de les réimplémenter à cette fin.

Les avantages des énumérations rapides sont nombreux, et c'est pour cela qu'Apple (et Sun avec Java) a suivi l'exemple donné par Python et C# :

- La syntaxe est plus compacte et le code bien plus lisible et compréhensible.
- Les performances sont meilleures qu'avec un énumérateur. Bien sûr, les performances dépendent des types de containers. Une liste est par exemple très appropriée pour un parcours linéaire, alors qu'un tableau est destiné à être accédé par un index.
- Il est interdit de modifier le container durant l'exécution de la boucle : une exception est levée dans ce cas. En conséquence, il est possible de faire plusieurs énumérations simultanément.

Implémenter les énumérations rapides pour vos propres classes

```
@interface Enumerable : NSObject
<NSFastEnumeration> {
    NSNumber * zero;
    NSNumber * un;
    NSNumber * deux;
}
@property (readwrite, assign) NSNumber * zero;
@property (readwrite, assign) NSNumber * un;
@property (readwrite, assign) NSNumber * deux;
@end
@implementation Enumerable
```

```
@synthesize zero;
@synthesize un;
@synthesize deux;
- (NSUInteger)countByEnumeratingWithState:
➥ (NSFastEnumerationState *)state objects:
➥ (id *)stackbuf count:(NSUInteger)len {
    NSUInteger fastEnumCount = 0;
    if (state->state == 0){
        stackbuf[0] = self.zero;
        fastEnumCount = 1;
        state->state = 1;
    }
    else if(state->state == 1){
        stackbuf[0] = self.un;
        fastEnumCount = 1;
        state->state = 2;
    }
    else if(state->state == 2){
        stackbuf[0] = self.deux;
        fastEnumCount = 1;
        state->state = 3;
    }
    else {
        return 0;
    }
    state->itemsPtr = stackbuf;
    state->mutationsPtr = (unsigned long *)self;
    return fastEnumCount;
}
@end
int exempleEnumerationImplementation(){
    Enumerable * enumerable = [[Enumerable alloc]
    ➥ init];
```

```
    enumerable.zero = [NSNumber numberWithInt:0];
    enumerable.un = [NSNumber numberWithInt:1];
    enumerable.deux = [NSNumber numberWithInt:2];
    for(id elem in enumerable){
        NSLog(@"élément: %@ de type %@», elem,
        ➥ [elem className]);
    }
}
```

L'exemple précédent retourne :

```
élément: 0 de type NSCFNumber
élément: 1 de type NSCFNumber
élément: 2 de type NSCFNumber
```

Pour que vos classes puissent être utilisées avec les énumérations rapides, il faut qu'elles implémentent le protocole `NSFastEnumeration`, qui ne contient qu'une seule méthode :`(NSUInteger)countByEnumeratingWithState:(NSFastEnumerationState *)state objects:(id *)stackbuf count:(NSUInteger)len`.

Malheureusement, ce n'est pas une méthode triviale à implémenter et il vous faudra de l'entraînement avant d'arriver à écrire votre première implémentation. Veuillez-vous reporter à la documentation d'Apple pour obtenir les informations nécessaires, comprendre ce que doit contenir l'instance de `NSFastEnumerationState` et ce que doit retourner votre méthode, dans les structures passées par référence.

De plus, il y a peu de classes qui nécessitent d'être parcourues de la sorte, donc il est fort probable que vous n'implémentiez ce protocole que très occasionnellement. Mais une fois ce protocole implémenté, il sera beaucoup plus simple d'itérer sur les instances de ladite classe.

L'exemple précédent montre également qu'il est possible d'itérer sur n'importe quel type de classe et de donnée, et qu'il peut être parfois astucieux de faire ainsi.

Que se passe-t-il quand nil reçoit un message ?

```
NSString * maChaineNil = nil;
NSNumber * longueur = [NSNumber numberWithInt:
➥ [maChaineNil length]];
NSLog(@»longueur: %@», longueur);
NSLog(@»description: %@»,
➥ [maChaineNil description]);
```

Ce code pourrait faire hurler n'importe quel développeur Java ou C#...

Pourtant, non seulement le code compile sans erreur, mais il s'exécute également sans erreur :

```
>> longueur: 0
>> description: (null)
```

La raison ? Contrairement à la plupart des autres langages de programmation, il est parfaitement valide d'envoyer un message à `nil`.

Dans les grandes lignes, la règle simplifiée à retenir est la suivante : quand un message est envoyé à `nil`, la valeur retournée est 0 si la méthode en question renvoyait un objet ou un objet de type scalaire (`float`, `int`, `double`, etc.)

Info

Nous avons choisi de simplifier ici la règle car celle-ci couvre 99 % des cas. Pour les cas non-traités que sont les méthodes retournant des structures ou des objets de type valeurs ne faisant pas partie des types primitifs d'Objective-C, nous vous invitons à consulter la référence Objective-C d'Apple.

Cette section ne se trouve pas par erreur dans le chapitre consacré à Objective-C 2.0. En effet, le comportement de l'environnement d'exécution d'Objective-C a changé entre la version 2.0 : dans la version 1.0 d'Objective-C un message envoyé à `nil` était valide à condition que la méthode retourne un objet. Dans ce cas, la valeur retournée était `nil`. Dans tous les autres cas, la valeur retournée était indéfinie.

4

Gestion des notifications et des événements

Dans ce chapitre, nous abordons le système de notifications utilisé en Objective-C. Dans le jargon Cocoa, les termes *événement* (*event*) et *notification* (*notification*) désignent deux concepts très proches, mais distincts, qui sont en général confondus dans par les autres langages de programmation sous le terme *événement* (par exemple, par Java ou C#).

Pour nous, le terme événement désigne une action provenant du matériel (souris, clavier, etc.), tandis que notification désigne le moyen par lequel un objet notifie les autres objets de l'occurrence d'un

événement logiciel. Ce système ce base sur le *modèle de conception* (*design pattern*) appelé *publication/abonnement* (*publish/subscribe*) qui fera l'objet de la première section.

Principe du modèle de conception publication/abonnement

Les développeurs C# et Java sont habitués à l'utilisation d'objets de type *événement* pour la gestion des événements : C# possède le mot-clé `event` permettant d'introduire un membre de type événement parmi les différentes variables d'instance de votre classe. Java permet d'obtenir le même résultat, mais en dérivant une nouvelle classe depuis `java.util.EventObject`. Le modèle de conception implémenté ici au niveau langage est le modèle *observateur/observable*.

Ce modèle consiste donc à mettre en relation directe les objets intéressés par les changements d'états d'un autre objet. On dit alors que les objets observateurs s'abonnent à certains événements de l'objet tiers (*observable*) qui prend alors à sa charge le rôle de les notifier lorsqu'une certaine propriété est modifiée ou lorsqu'un certain événement se produit.

Objective-C propose une généralisation de cette approche, appelée le modèle de conception *publication/abonnement* : ce ne sont pas les ici les observables

(objets observés) qui prennent à leur charge d'informer les observateurs, mais un intermédiaire appelé centre de notifications qui, dans le cas de Cocoa est une instance de la classe `NSNotificationCenter`.

Un centre de notifications est donc tout simplement un objet qui prend à sa charge la transmission des notifications depuis l'objet émetteur vers l'objet observateur. Une analogie serait par exemple Tweeter : vous (objet observé) envoyez un *tweet* (notification) à Tweeter.com qui occupe le rôle du centre de notifications qui va l'acheminer vers le destinataire (observateur) en leur envoyant, par exemple, un SMS.

Cette implémentation est plus générique et découple totalement les différents objets : un objet éditeur envoie ses événements au centre de notifications et n'a donc besoin de rien savoir a propos de ses abonnés, et les abonnés ne communiquent qu'avec le centre de notifications.

Les principaux avantages résultant de ce découplage sont qu'un objet envoyant une notification n'est pas en charge de savoir si des observateurs sont abonnés, et de plus le centre de notifications faisant office d'intermédiaire, l'objet n'a pas non plus besoin de savoir si les objets sont locaux ou pas. Il est ainsi possible d'abstraire les communications distantes et de créer des systèmes distribués très facilement.

Obtenir le centre de notifications par défaut

```
[NSNotificationCenter defaultCenter]
```

Chaque processus dispose d'un centre de notifications par défaut, et vous n'avez pas à créer d'instance de cette classe. Vous obtenez l'instance par défaut en envoyant le message `defaultCenter` à l'objet-classe `NSNotificationCenter`.

Comme nous l'avons vu dans la section précédente, il est possible de créer de manière transparente un centre de notifications distribué. `NSNotificationCenter` n'envoie les notifications que dans le processus en cours d'exécution. Mais il existe également `NSDistributedNotificationCenter` qui permet d'envoyer les notifications vers différents processus (uniquement en local dans l'implémentation actuelle). Les notifications distribuées sont beaucoup plus lourdes, il ne faut donc pas de les utiliser par défaut.

Les notifications distribuées sont très importantes pour le développement d'applications distribuées. Toutefois, `NSDistributedNotificationCenter` et les sujets relatifs sortent du cadre de cet ouvrage et nous vous recommandons de vous reporter à la documentation d'Apple pour les découvrir.

Poster une notification synchrone

```
    [[NSNotificationCenter defaultCenter]
    ➥ postNotification:[NSNotification
    ➥ notificationWithName:@"maNotification"
    ➥ object:instance]];
//équivalent à :
NSNotification * notif = [NSNotification
➥ notificationWithName:@"maNotification"
➥ object:instance]
[[NSNotificationCenter defaultCenter]
➥ postNotification:notif];
//équivalent à :
[[NSNotificationCenter defaultCenter]
➥ postNotification:@"maNotification"
➥ object:instance]];
```

Comme vous pouvez le voir, il est assez simple d'envoyer une notification synchrone : il suffit de créer une nouvelle instance de `NSNotification` et de l'envoyer au centre de notifications par défaut.

Il est également possible d'envoyer directement le message `postNotification` au centre par défaut avec les arguments nécessaires à la création de l'instance de `NSNotification`, et le centre de notifications gérera le reste.

Il existe trois versions de la méthode `postNotification`, deux d'entre-elles étant des versions simplifiées de la principale :

```
- postNotification:  //version principale
                     //NSNotification comme argument
- postNotificationName:object: //userInfo est null
- postNotificationName:object:userInfo:
```

Voici un autre exemple montrant l'utilisation de `postNotificationName:object:userInfo:` et donc également comment il est possible de passer des informations supplémentaires sous la forme du dictionnaire `userInfo` :

```
NSDictionary* dic = [NSDictionary
➥ dictionaryWithObjectsAndKeys: @"clef1",
➥ [NSNumber numberWithInt:1], @"clef2",
➥ [NSNumber numberWithInt:2], @"clef3",
➥ [NSNumber numberWithInt:3], nil];
[[NSNotificationCenter defaultCenter]
➥ addObserver:instance
➥ selector:@selector(afficherNotification:)
➥ name:@"maNotification" object:instance];
[[NSNotificationCenter defaultCenter]
➥ postNotificationName:@"maNotification"
➥ object:instance userInfo:dic];
```

Les notifications envoyées ainsi au centre de notifications sont synchrones : la méthode `postNotification:` ne rend la main qu'une fois toutes

les notifications envoyées et que les observateurs ont traité la notification. C'est donc une manière de procéder peu performante.

Poster une notification asynchrone

```
NSNotification * notification =  [NSNotification
➥ notificationWithName:@"maNotification"
➥ object:instance userInfo:dic];
[[NSNotificationQueue defaultQueue]
➥ enqueueNotification:notification
➥ postingStyle:NSPostASAP];
```

Comme nous l'avons vu à la section précédente, les envois de notifications au centre par défaut *via* les différentes méthodes `postNotification:` sont synchrones. Il est possible d'envoyer les notifications de manière asynchrone, *via* l'utilisation des files d'attente de notifications, instances de la classe `NSNotificationQueue`. Nous verrons dans la section suivante que ces files fournissent une seconde fonctionnalité très importante.

La méthode de classe `defaultQueue` retourne la file d'attente par défaut du centre de notifications par défaut.

Info

Dans l'exemple précédent, nous avons créé une instance de `NSNotification` que nous passons à la file d'attente par défaut *via* la fonction `enqueueNotification:postingStyle:`. Nous avons dissocié l'appel en deux étapes afin d'améliorer la lisibilité du code dans l'ouvrage. En général, les développeurs Objective-C préfèrent les appels imbriqués.

Il existe trois différentes manières d'envoyer la notification, qui sont définies dans le fichier `NSNotification.h` sous la forme d'une énumération :

```
//NSNotificationQueue.h
enum {
    NSPostWhenIdle = 1,
    NSPostASAP = 2,
    NSPostNow = 3
};
```

- Comme son nom l'indique `NSPostNow` signifie que la notification doit être envoyée immédiatement. La notification ne sera donc *pas* asynchrone, mais bénéficiera toutefois de la fonction de *regroupement* (*coalescing*) des notifications de la file d'attente.Vous la découvrirez à la section suivante.
- Lorsque vous avez besoin d'accéder à une ressource rare ou chère, et que vous souhaitez toutefois que votre notification soit envoyée dès que

possible, vous utiliserez le style `NSPostASAP` (*As Soon As Possible*, dès que possible). Cette méthode est asynchrone et retourne immédiatement.

- Enfin, `NSPostWhenIdle` propose d'envoyer la notification lorsque la file d'attente est en attente de nouvelles notifications. Cela signifie que `NSPostWhenIdle` est une manière d'envoyer les notifications avec une priorité minimale. Cette méthode est asynchrone et retourne immédiatement.

Info

Chaque thread dispose d'une file d'attente par défaut, mais il est possible de créer vos propres files d'attente et les associer au centre de notifications et à vos thread. Ceci est une utilisation avancée qui ne sera pas traitée dans ici, et nous vous renvoyons donc à la documentation d'Apple.

Il est possible de retirer une notification d'une file d'attente grâce à la méthode `dequeueNotifications-Matching:coalesceMask:` de `NSNotificationQueue`. Son fonctionnement est similaire à celui de la méthode `removeObserver:name:object:` de `NSNotificationCenter`, mais contrairement à celle-ci, son utilisation reste très rare.

Regrouper par nom ou par émetteur (et supprimer) les notifications d'une file d'attente

```
NSNotification * notification = [NSNotification
➥ notificationWithName:@"maNotification"
➥ object:instance userInfo:dic];
[[NSNotificationQueue defaultQueue]
➥ enqueueNotification:notification
➥ postingStyle:NSPostNow
➥ coalesceMask:NSNotificationNoCoalescing
➥ forModes:nil];
[[NSNotificationQueue defaultQueue]
➥ enqueueNotification:notification
➥ postingStyle:NSPostASAP coalesceMask:
➥ NSNotificationCoalescingOnName forModes:nil];
[[NSNotificationQueue defaultQueue]
➥ enqueueNotification:notification
➥ postingStyle:NSPostWhenIdle
➥ coalesceMask:NSNotificationCoalescingOnSender
➥ forModes:nil];
```

La méthode `enqueueNotification:postingStyle:coalesceMask:forModes:` de `NSNotificationQueue` permet, *via* le paramètre `coalesceMask`, de regrouper les notifications similaires dans la file d'attente avant que celles-ci ne soit envoyées.

La seconde fonctionnalité apportée par les files d'attente est la possibilité de regrouper les notifications similaires avant qu'elles ne soient envoyées au

centre de notifications. Cela permet bien évidemment d'optimiser les envois et donc le traitement des notifications en supprimant par exemple les notifications en double dans la file d'attente. Dans 99 % des cas, il faut passer `nil` à `forModes:` afin de signaler qu'il faut utiliser le mode par défaut, c'est-à-dire `NSDefaultRunLoopMode`.

Il existe trois – en fait, deux, la troisième stipulant de ne pas regrouper – différentes manières de regrouper les notifications, qui sont définies dans le fichier `NSNotification.h` sous la forme d'une énumération :

```
// NSNotificationQueue.h
enum {
    NSNotificationNoCoalescing = 0,
    NSNotificationCoalescingOnName = 1,
    NSNotificationCoalescingOnSender = 2
};
```

- Comme son nom l'indique, `NSNotificationNoCoalescing` indique de pas regrouper les notifications.
- Afin de regrouper les notifications qui portent le même nom, vous utiliserez `NSNotificationCoalescingOnName`. Cela signifie que toutes les notifications de la file portant le même nom que la notification que vous êtes en train d'envoyer vont être supprimées et remplacées par cette dernière.

- Afin de regrouper les notifications qui proviennent du même émetteur, vous utiliserez `NSNotificationCoalescingOnSender`. Cela signifie que toutes les notifications de la file provenant du même objet que celui qui est en train d'envoyer une nouvelle notification vont être supprimées.

Regrouper par nom et par émetteur (et supprimer) les notifications d'une file d'attente

```
NSNotification * notification =
➥ [NSNotification notificationWithName:
➥@"maNotification" object:instance userInfo:dic];
[[NSNotificationQueue defaultQueue]
➥ enqueueNotification:notification postingStyle:
➥ NSPostWhenIdle coalesceMask:
➥ NSNotificationCoalescingOnName |
➥ NSNotificationCoalescingOnSender forModes:nil];
```

Nous avons vu à la section précédente comment regrouper et supprimer les notifications d'une file d'attente portant le même nom ou provenant du même émetteur. Toutefois, cette manière de procéder peut être trop ouverte et il est plus courant de regrouper les notifications portant le même nom *et* provenant du même objet.

L'exemple ci-dessus montre comment utiliser l'opérateur ou-binaire (la barre verticale "`|`") à cette fin.

S'abonner pour recevoir les notifications

```
[[NSNotificationCenter defaultCenter] addObserver:
➥ instance selector:@selector(afficherNotification:)
➥ name:@"maNotification" object:instance];
```

Un objet peut s'abonner auprès du centre de notifications grâce à la méthode `addObserver:selector:name:object:`. Ainsi, l'exemple précédent ajoute l'objet `instance` aux abonnés de la notification `maNotification` provenant de l'objet `instance`. À chaque fois que l'objet `instance` enverra l'événement `maNotification` au centre de notifications, celui-ci enverra à instance le message `afficherNotification:` – auquel il a accès grâce au sélecteur (de type `SEL`, créé avec `@selector`) passé en argument.

Voici un exemple trivial de la méthode `afficherNotification:`. Vous pourrez noter au passage l'utilisation de la Dot Syntax.

```
- (void) afficherNotification:(NSNotification *)
  ➥ notif{
    NSLog(@"Notification %@ envoyé par %@ et info
    ➥%@", notif.name, notif.object,
    ➥ notif.userInfo);
}
```

Il est possible de s'abonner à tous les événements d'un objet en passant `nil` comme paramètre pour

name:, de même qu'il est possible de s'abonner à tous les événements portant un certain nom, quel que soit l'objet en passant nil comme paramètre pour object:.

Comme vous avez pu vous en rendre compte dans cet exemple, il est tout à fait possible pour un objet de s'abonner à ses propres événements.

Attention

D'après ce que nous venons de voir, vous vous demandez sans doute ce qu'il se passe si nil est passé comme paramètre de name: et d'object: ? Il se passe exactement ce que vous pensez : vous êtes abonné à toutes les notifications du centre de notifications, c'est-à-dire que toutes les notifications de votre application vont être envoyées à votre instance. Bien évidemment, il est fortement déconseillé de procéder de la sorte, vu l'impact négatif en termes de performance que vous allez subir.

Annuler un abonnement

```
[[NSNotificationCenter defaultCenter]
➥ removeObserver:instance];  //cas 1
[[NSNotificationCenter defaultCenter]
➥ removeObserver: instance name:@"maNotification"
➥ object:instance];   //cas 2
```

Vous annulez l'abonnement d'un objet en envoyant le message `removeObserver:` ou `removeObserver:name:object:` au centre de notifications.

Pour annuler tous les abonnements d'un certain objet, vous utiliserez `removeObserver:` en passant l'instance dont les abonnements doivent être supprimés du centre de notifications (cas 1).

Pour annuler un abonnement spécifique d'un objet spécifique, vous enverrez le message `removeObserver:name:object:` (cas 2).

De même qu'il est possible de s'abonner à toutes les notifications d'un objet ou un certain type de notification quel que soit l'objet, il est également possible de supprimer de manière sélective les abonnements. Vous pouvez annuler les différents abonnements vers une certaine notification en passant `nil` pour `object:` et vous pouvez annuler toutes les différentes notifications en provenance d'un objet en passant `nil` comme `name:`.

Attention

Le centre de notifications ne garde qu'une *référence faible* vers les objets abonnés. Il ne faut donc surtout pas oublier de retirer vos objets du centre de notifications avant de les libérer. Le centre de notifications n'ayant pas de références fortes vers ses abonnés, les objets sont libérés par le ramasse-miettes ou par `dealloc` sans que le centre ne supprime la référence qu'il détient. Lorsque cela se produit et qu'une notification doit être envoyée par le centre vers un objet qui n'existe pas, votre programme plante.

Les différents types d'événements

Nous avons vu dans l'introduction de ce chapitre que le terme *événement* désigne, dans le jargon Cocoa Objective-C, une action provenant du matériel (souris, clavier, etc.). Ceci est différent de .Net C# et Java, où les événements (`event`*s*) sont équivalents aux notifications (`NSNotifications`) de Cocoa Objective-C.

La classe `NSEvent` définit les événements matériel et l'énumération `NSEventType` l'ensemble des événements gérés par le système (27 différents lors de la rédaction de ce texte). Apple classifie ces événements en six grandes catégories, qui sont :

- souris : clics et mouvements ;
- clavier ;
- zone de suivi : le pointeur de la souris entre/sort/glisse dans une zone prédéfinie ;
- tablette graphique ;
- événements périodiques (par exemple, lorsque l'utilisateur maintient une touche du clavier enfoncée, le système répète de manière périodique le bouton, avec la fréquence définie dans les Préférences Système) ;
- autres.

Notez que la très grande majorité des événements est générée par la souris et le clavier. Nous nous

contenterons donc de voir la gestion de ces derniers au cours de ce texte. La gestion des autres événements est toutefois similaire et nous vous renvoyons aux exemples et à la documentation d'Apple.

À noter que les zones de suivi permettent d'optimiser la gestion des mouvements de la souris en limitant la zone où les événements générés par ces derniers sont transférés au gestionnaire d'événements.

Les événements étant des actions provenant du matériel, la classe `NSEvent` fait partie de la bibliothèque `AppKit` de Cocoa, regroupant l'ensemble des éléments nécessaires à la création et à la gestion des interfaces utilisateurs.

Fonctionnement de la gestion des événements

Lorsque vous lancez une application Cocoa, le système crée une instance de `NSApplication`, ainsi qu'une variable globale, `NSApp`, qui référence cette instance.

Au même moment, une source d'événements (*event source*) est créée au niveau du système d'exploitation.

Le gestionnaire de fenêtres (*window server*) de Mac OS X reçoit les événements matériels (par exemple, un clic de souris ou des actions sur le clavier)

et s'occupe de les placer dans la source d'événements adéquate afin qu'ils soient traités.

Chaque application Cocoa dispose d'une boucle d'exécution principale (*main run loop*) qui est une instance de `NSRunLoop` lié au thread principal (*main thread*). La boucle d'exécution principale est chargée d'aller chercher les événements dans cette source d'événements.

`NSApp` obtient les événements placés dans la source et les convertit en instances de la classe `NSEvent` qu'il va ensuite rediriger vers les classes appropriées afin qu'elles soient traitées.

Par exemple, pour un clic de souris, `NSApp` transfère l'instance d'`NSEvent` vers la fenêtre (instance de `NSWindow`) qui a reçu le clic, qui va lui-même la transférer vers la vue correspondante (instance de `NSView`).

Pour un touche du clavier, `NSApp` va transférer l'instance d'`NSEvent` vers le premier répondeur (*first responder*) de la fenêtre actuellement au premier plan (*key window*).

Dans tous les cas, chaque classe décidera ou non de traiter l'événement, et s'il ne le traite pas, l'événement sera renvoyé vers le prochain répondeur. L'ordre dans lequel l'événement parcourt les différents répondeurs est déterminé par la hiérarchie des différents éléments d'interface et porte le nom de chaîne de répondeurs (*respondeur chain*).

Enfin, à noter que lorsqu'une instance de `NSRepondeur` reçoit un événement, il peut le convertir en message d'action qui sera traité par la classe désignée pour cette action (*via* InterfaceBuilder en général). Par exemple, les événements créés à la suite d'un clic sur le bouton de fermeture de fenêtre et du raccourci clavier Pomme-W seraient tous deux convertis vers le même message d'action : la méthode `close` de la classe `NSWindow`.

Info

Nous venons de voir une version très simplifiée de ce qui se passe entre le moment où le gestionnaire de fenêtres reçoit un événement, le transmet à une application Cocoa et le moment où l'événement est traité.

Pour bien utiliser Cocoa, il est très important de bien comprendre le fonctionnement et la gestion des événements.

Nous vous renvoyons donc à la référence Cocoa d'Apple qui est riche en information à ce sujet.

En général, lorsque vous avez besoin de capturer directement les événements de la souris, c'est que vous avez besoin de définir un comportement particulier (par exemple, vous développez un logiciel de dessin) ou que vous être en train de définir un élément d'interface (par exemple, un nouveau bouton ne faisant pas partie de la palette standard).

En conséquence, vous allez créer une nouvelle sous-classe de `NSView` (ou de `NSControl` qui dérive de `NSView`).

Attention

Il est important de noter qu'il n'est en général pas nécessaire de capturer les événements lorsque vous utilisez les éléments de la palette standard tels que `NSButton`. En effet, ces classes prennent à leur charge la gestion des événements et transmettent des messages d'action (*action messages*) vers l'application lorsque nécessaire (par exemple, lorsqu'un clic ou un double-clic a été détecté).

La gestion des événements se fait au niveau de la classe `NSResponder`, dont dérive `NSView`. Gérer ces événements revient donc tout simplement à implémenter les méthodes correspondantes de `NSResponder` qui définit, entre autres, les événements suivants issus de la souris :

```
- mouseDown:          //bouton gauche de la
                      //souris enfoncé
- mouseDragged:       //souris déplacée avec le
                      //bouton gauche enfoncé
- mouseUp:            //bouton gauche relâché
                      //après avoir été appuyé
- mouseMoved:         //souris déplacée
- mouseEntered:       //la souris entre dans une
                      //zone prédéfinie
```

```
- mouseExited:        //la souris sort de la
                      //zone prédéfinie
- rightMouseDown:     //idem que mouseDown mais
                      //avec le bouton droit
- rightMouseDragged: //idem que mouseDragged
                      //mais avec le bouton droit
- rightMouseUp:       //idem que mouseUp mais
                      //avec le bouton droit
- otherMouseDown:     //idem que mouseDown mais
                      //pour le bouton du milieu
- otherMouseDragged: //idem que mouseDragged mais
                      //pour le bouton du milieu
- otherMouseUp:       //idem que mouseUp mais
                      //pour le bouton du milieu
```

Et les événements suivants i ssus du clavier :

```
- keyDown:      //touche appuyée
- keyUp:        //touche relâchée
- flagsChanged: //une ou plusieurs touches
                //spéciales appuyées ou relâchées
```

Gérer les événements provenant de la souris

```
enum {        /* various types of events */
    NSLeftMouseDown             = 1,
    NSLeftMouseUp               = 2,
    NSRightMouseDown            = 3,
    NSRightMouseUp              = 4,
```

```
    NSMouseMoved                = 5,
    NSLeftMouseDragged          = 6,
    NSRightMouseDragged         = 7,
    NSMouseEntered              = 8,
    NSMouseExited               = 9,
// […]
    NSOtherMouseUp              = 26,
    NSOtherMouseDragged         = 27
};
```

Le fichier `NSEvent.h` définit les événements souris ci-dessus. Nous allons voir les différents événements émis par la souris et comment gérer le plus simple : le clic de souris.

- `NSLeftMouseDown` /`NSRightMouseDown` : bouton gauche/droit de la souris enfoncée.
- `NSLeftMouseUp` / `NSRightMouseUp` : bouton gauche/droit de la souris relevée après appui.
- `NSMouseMoved`/`NSLeftMouseDragged`/`NSRightMouseDragged` : la souris a été déplacée sans bouton maintenu enfoncé/avec le bouton gauche maintenu enfoncé/avec le bouton droit maintenu enfoncé.
- `NSMouseEntered` / `NSMouseExited` : la souris est entrée dans la zone.
- `NSOtherMouseUp` / `NSOtherMouseDragged` : même chose avec le troisième (ou supérieur) bouton de la souris.

Le clic de la souris correspond à l'événement `mouseUp:`

Vous l'aviez sans doute déjà remarqué, mais il est important de le rappeler : le clic n'est pas validé tant que vous n'avez pas relâché le bouton de la souris, et n'est donc pas pris en compte par l'événement **`mouseDown:`** mais par l'événement **`mouseUp:`** (sans doute pour permettre à l'utilisateur de valider sa décision et lui donner l'opportunité de changer d'avis entre le moment où il appuie sur le bouton et celui où il va le relâcher).

```
- (void) afficherEvenement:(NSEvent*) event {
    switch([event type]){
        case NSRightMouseDown:
            NSLog(@"Bouton droit appuyé: %@",
            ➥ [event description]);
            break;
        case NSRightMouseUp:
            NSLog(@"Clic droit: %@",
            ➥ [event description]);
            break;
        case NSLeftMouseDown:
            NSLog(@"Bouton gauche appuyé: %@",
            ➥ [event description]);
            break;
        case NSLeftMouseUp:
            NSLog(@"Clic gauche: %@",
            ➥ [event description]);
```

```
            break;
        default:
            NSLog(@"Autre événement : %@",
            ➥ [event description]);
    }
}
- (void) mouseDown:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) rightMouseDown:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) mouseUp:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) rightMouseUp:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) otherMouseDown:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) otherMouseUp:(NSEvent*)event {
    [self afficherEvenement:event];
}
```

Voici ce que vous obtenez avec le code précédent :

```
Bouton gauche appuyé: NSEvent: type=LMouseDown
➥ loc=(284,280) time=202539.3 flags=0x100 win=0x0
➥ winNum=246200 ctxt=0x12bc7 evNum=11997 click=1
➥ buttonNumber=0 pressure=1
```

```
Clic gauche: NSEvent: type=LMouseUp loc=(284,280)
➥ time=202539.4 flags=0x100 win=0x0 winNum=246200
➥ ctxt=0x12bc7 evNum=11997 click=1 buttonNumber=
➥ 0pressure=0
Bouton droit appuyé: NSEvent: type=RMouseDown
➥ loc=(351,271) time=202783.5 flags=0x100 win=0x0
➥ winNum=246200 ctxt=0x12bc7 evNum=12042 click=1
➥ buttonNumber=1 pressure=0
Clic droit: NSEvent: type=RMouseUp loc=(351,271)
➥ time=202783.6 flags=0x100 win=0x0 winNum=246200
➥ ctxt=0x12bc7 evNum=12042 click=1 buttonNumber=1
➥ pressure=0
Autre  événement : NSEvent: type=OtherMouseDown
➥ loc=(351,271) time=202784.6 flags=0x100 win=0x0
➥ winNum=246200 ctxt=0x12bc7 evNum=86 click=1
➥ buttonNumber=2 pressure=1
Autre  événement : NSEvent: type=OtherMouseUp
➥ loc=(351,271) time=202784.8 flags=0x100 win=0x0
➥ winNum=246200 ctxt=0x12bc7 evNum=86 click=1
➥ buttonNumber=2 pressure=0
```

Il est alors très facile d'obtenir toute information utile depuis l'instance de `NSEvent` passée à la méthode, telle que l'endroit où le clic est survenu (avec le message `locationInWindow`), le nombre de clics (avec le message `clickCount`), etc. Veuillez-vous référer à la documentation de `NSEvent` pour découvrir l'ensemble des méthodes implémentées par cette classe.

Gérer les événements provenant du clavier

```
- keyDown:
- keyUp:
- flagsChanged:
```

La gestion des appuis sur les touches du clavier se fait de la même manière que celles des clics de la souris, en définissant les méthodes de `NSController` dans votre sous-classe, principalement.

Accepter explicitement de recevoir les événements du clavier

Par défaut, **`NSView`** ne gère pas les événements issus du clavier. Votre instance de **`NSView`** ne va donc les recevoir que si elle accepte explicitement de les recevoir. Dans le cas contraire, l'événement est transféré au répondeur suivant dans la chaîne des répondeurs. C'est pourquoi nous implémentons **`acceptFirstReponder`** dans l'exemple suivant.

Pour plus d'informations au sujet de la chaîne des répondeurs, veuillez-vous reporter à la documentation d'Apple.

```
- (void) afficherEvenement:(NSEvent*) event {
    switch([event type]){
        case NSKeyUp:
            NSLog(@"Touche relâchée: %@",
            ➥ [event description]);
            break;
        case NSKeyDown:
            NSLog(@"Touche appuyée: %@",
            ➥ [event description]);
            break;
        case NSFlagsChanged:
            NSLog(@"Touches spéciales changées: %@",
            ➥ [event description]);
            break;
        default:
            NSLog(@"Autre  événement : %@",
            ➥ [event description]);
    }
}
- (void) keyDown:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) keyUp:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (void) flagsChanged:(NSEvent*)event {
    [self afficherEvenement:event];
}
- (BOOL) acceptsFirstResponder{
    NSLog(@"Accepting first responder");
    return YES;
}
```

L'exemple précédent génère la sortie suivante :

```
Accepting first responder
Touche appuyée: NSEvent: type=KeyDown loc=(0,499)
➥ time=204506.4 flags=0x100 win=0x0 winNum=248437
➥ ctxt=0x235c3 chars="a" unmodchars="a" repeat=0
➥ keyCode=12
Touche relâchée: NSEvent: type=KeyUp loc=(0,499)
➥ time=204506.5 flags=0x100 win=0x0 winNum=248437
➥ ctxt=0x235c3 chars="a" unmodchars="a" repeat=0
➥ keyCode=12
Touche appuyée: NSEvent: type=KeyDown loc=(0,499)
➥ time=204506.5 flags=0x100 win=0x0 winNum=248437
➥ ctxt=0x235c3 chars="z" unmodchars="z" repeat=0
➥ keyCode=13
Touche relâchée: NSEvent: type=KeyUp loc=(0,499)
➥ time=204506.7 flags=0x100 win=0x0 winNum=248437
➥ ctxt=0x235c3 chars="z" unmodchars="z" repeat=0
➥ keyCode=13
Touches spéciales changées: NSEvent:
➥ type=FlagsChanged loc=(0,499) time=204511.7
➥ flags=0x100110 win=0x0 winNum=248437 ctxt=0x235c3
➥ keyCode=54
Touches spéciales changées: NSEvent:
➥ type=FlagsChanged loc=(0,499) time=204511.8
➥ flags=0x100 win=0x0 winNum=248437 ctxt=0x235c3
➥ keyCode=54
```

Gérer les touches spéciales et les combinaisons

```
enum {              /* masks for the types of events */
    NSLeftMouseDownMask        = 1 << NSLeftMouseDown,
    NSLeftMouseUpMask          = 1 << NSLeftMouseUp,
    NSRightMouseDownMask       = 1 << NSRightMouseDown,
    NSRightMouseUpMask         = 1 << NSRightMouseUp,
    NSMouseMovedMask           = 1 << NSMouseMoved,
    NSLeftMouseDraggedMask     = 1 <<
                               ➥ NSLeftMouseDragged,
    NSRightMouseDraggedMask    = 1 <<
                               ➥ NSRightMouseDragged,
    NSMouseEnteredMask         = 1 << NSMouseEntered,
    NSMouseExitedMask          = 1 << NSMouseExited,
    NSKeyDownMask              = 1 << NSKeyDown,
    NSKeyUpMask                = 1 << NSKeyUp,
    NSFlagsChangedMask         = 1 << NSFlagsChanged,
    //[…]
    NSAnyEventMask             = NSUIntegerMax
};
enum {
    NSAlphaShiftKeyMask        = 1 << 16,
                      ➥//touche verrouillage majuscule
    NSShiftKeyMask       = 1 << 17, //touche majuscule
    NSControlKeyMask     = 1 << 18, //touche Contrôle
    NSAlternateKeyMask   = 1 << 19, //touche Alt aussi
                                    //appelée option
    NSCommandKeyMask     = 1 << 20, //touche Commande
                                    //aussi appelée
                                    //Pomme
    // […]
};
```

La gestion des touches spéciales et des combinaisons se fait grâce aux masques définis dans `NSEvent.h`.

Les touches spéciales sont les touches Maj, Ctrl, Option, Pomme, etc., tandis que les combinaisons incluent deux groupent distincts :

- Les combinaisons de touches (servant aux raccourcis clavier, mais également à certains caractères dans les alphabets complexes ou même les lettres accentuées simples sur les claviers américains par exemple).
- Les combinaisons à la souris et au clavier (comme par exemple Pomme+clic).

Voici, par exemple, comment nous modifierions l'implémentation de notre méthode `afficherEvenement:` afin de reconnaître la combinaison Pomme+clic :

```
- (void) afficherEvenement:(NSEvent*) event {
    switch([event type]){
        case NSRightMouseDown:
            NSLog(@"Bouton droit appuyé: %@",
            ➥ [event description]);
            break;
        case NSRightMouseUp:
            NSLog(@"Clic droit: %@",
            ➥ [event description]);
            break;
        case NSLeftMouseDown:
            NSLog(@"Bouton gauche appuyé: %@",
            ➥ [event description]);
```

```
                break;
            case NSLeftMouseUp:
                if ([event modifierFlags] &
                ➥ NSCommandKeyMask){
                    NSLog(@"Pomme Clic: %@",
                    ➥ [event description]);
                }
                else {
                    NSLog(@"Clic gauche: %@",
                    ➥ [event description]);
                }
                break;
            //[…]
            default:
                NSLog(@"Autre événement : %@",
                ➥ [event description]);
        }
}
```

Il est alors très facile de voir l'ensemble des événements successifs survenant lors d'une combinaison Pomme+clic :

```
Touches spéciales changées: NSEvent:
➥ type=FlagsChanged loc=(0,499) time=206323.3
➥ flags=0x100108 win=0x0 winNum=252899
➥ ctxt=0x1e98b keyCode=55
Bouton gauche appuyé: NSEvent: type=LMouseDown
➥ loc=(247,304) time=206324.1 flags=0x100108
➥ win=0x0 winNum=252899 ctxt=0x1e98b evNum=12387
➥ click=1 buttonNumber=0 pressure=1
```

```
Pomme Clic: NSEvent: type=LMouseUp loc=(247,304)
➥ time=206324.2 flags=0x100108 win=0x0
➥ winNum=252899 ctxt=0x1e98b evNum=12387 click=1
➥ buttonNumber=0 pressure=0
Touches spéciales changées: NSEvent:
➥ type=FlagsChanged loc=(0,499) time=206326.6
➥ flags=0x100 win=0x0 winNum=252899 ctxt=0x1e98b
➥ keyCode=55
```

Reconnaître les touches fonctionnelles

```
- (void) keyDown:(NSEvent*)event {
    [self afficherEvenement:event];
    NSString * chaine = [event characters];
    if([chaine length] == 1){
        unichar caractere = [chaine
        ➥ characterAtIndex:0];
        if (caractere == NSRightArrowFunctionKey) {
            NSLog(@"Flèche droite appuyée");
        }
        else if (caractere ==
        ➥ NSLeftArrowFunctionKey) {
            NSLog(@"Flèche gauche appuyée");
        }
        else if (caractere ==
        ➥ NSUpArrowFunctionKey) {
            NSLog(@"Flèche haute appuyée");
        }
```

```
        else if (caractere ==
        ➥ NSDownArrowFunctionKey) {
         NSLog(@"Flèche basse appuyée");
        }
        else {
            NSLog(@"autre touche");
        }
    }
}
```

Le code ci-dessus montre comment modifier notre méthode `keyDown:` des sections précédentes afin de tester les caractères correspondant à l'événement et de savoir si les touches fléchées ont été appuyées.

Les touches fonctionnelles (*function keys*) représentent les touches du clavier autres que les touches de caractères, tels que les touches les touches fléchées, les touches F1 à F12, etc.

Elles sont définies dans le fichier `NSEvent.h` :

```
enum {
    NSUpArrowFunctionKey        = 0xF700,
    NSDownArrowFunctionKey      = 0xF701,
    NSLeftArrowFunctionKey      = 0xF702,
    NSRightArrowFunctionKey     = 0xF703,
    NSF1FunctionKey             = 0xF704,
    //[…]
    NSF35FunctionKey            = 0xF726,
    NSInsertFunctionKey         = 0xF727,
    NSDeleteFunctionKey         = 0xF728,
```

```
    NSHomeFunctionKey           = 0xF729,
    NSBeginFunctionKey          = 0xF72A,
    NSEndFunctionKey            = 0xF72B,
    NSPageUpFunctionKey         = 0xF72C,
    NSPageDownFunctionKey       = 0xF72D,
    //[…]
};
```

Apple a fait correspondre un caractère Unicode à chacune de ces touches. Donc même si un appui sur ces touches ne fait pas apparaître de caractères à l'écran, leur gestion en Objective-C passe par `NSString` et `unichar`.

La méthode `characters` de la classe `NSEvent` retourne les caractères associés lors des événements `keyUp:` et `keyDown:`, tandis que la méthode `charactersIgnoringModifers` retourne les caractères associés à l'événement, mais en supprimant les modifications apportées par les touches spéciales (par exemple la combinaison Majuscule A retournera la lettre minuscule a).

Dans l'exemple ci-dessus, nous souhaitons savoir si une touche fléchée seule a été appuyée, et non pas une combinaison de touches incluant les touches fléchées. Nous extrayons donc la chaîne de caractères associé à l'événement, et, si la longueur de cette dernière est de 1 caractère seulement, nous extrayons ce caractère afin de le comparer aux constantes définies dans le fichier `NSEvent.h`.

5

Qualité du code

Nous abordons ici la qualité du code au sens large. Comment faire en sorte que le code soit stable ? Comment faire de sorte que les erreurs soient correctement propagées ? Comment et où gérer ces erreurs ? Comment tester son code, s'assurer qu'il fonctionne bien et éviter les régressions ?

À la fin de ce chapitre, vous devriez être en mesure d'écrire du code stable, testé, performant tout en gérant les erreurs en suivant les conventions et recommandations d'Apple bref, de produire du code de qualité ! Notez toutefois que performant ne signifie par optimisé. L'optimisation est l'étape ultime à n'aborder qu'une fois que vous avez produit du code juste et de qualité.

Activer le support des exceptions dans Objective-C

Le support des exceptions au niveau du langage Objective-C et du compilateur GCC date de l'introduction de Mac OS X 10.3 (et GCC 3.3). C'est pourquoi, vous pouvez activer (et désactiver) les exceptions, afin de gérer le niveau de rétrocompatibilité dont vous avez besoin (voir Figure 5.1).

Figure 5.1 : Activation des exceptions depuis l'Inspecteur d'Xcode.

L'activation des exceptions se fait *via* le drapeau `-fobjc-exceptions` du compilateur.

Si vous utilisez Xcode, la manière la plus simple de procéder consiste à utiliser l'inspecteur de projet comme indiqué à la Figure 5.1.

De plus, le support des exceptions est activée par défaut dans Xcode.Vous n'utiliserez donc l'inspecteur que dans les rares cas où vous devez compiler du code pour Mac OS X 10.2 et précédents.

Lever une exception

```
@throw [NSException exceptionWithName:
➥@"ExceptionGuideDeSurvie" reason:@"À titre
➥d'exemple" userInfo:nil];
```

Tout comme en Java ou C#, où le mot-clé `throw` est utilisé pour lancer une exception, Objective-C propose la directive `@throw` suivie de l'objet à lancer.

Toutefois, à la différence de Java et C#, où l'objet lancé doit dériver respectivement des classes `Throwable` et `Exception`, Objective-C se comporte davantage comme Python ou C++ où il est possible de lancer un objet de n'importe quel classe.

Il est toutefois fortement déconseiller de lancer des objets ne dérivant pas de la classe `NSException`, car leur gestion devient très difficile et peuvent même

mener à des plantages de vos applications car Cocoa ne capture que les instances de `NSException` et ses dérivés.

Enfin, il est également possible de lever une exception sans passer par la directive `@throw` en utilisant la méthode d'instance `raise` ou la méthode de classe `raise:format:arguments:` de `NSException`.

```
@throw @"instance de NSString";  //1: possible,
//mais fortement déconseillé!
@throw [NSNumber numberWithInt:12];
//2: possible, mais fortement déconseillé!
@throw [NSException
➥ exceptionWithName:@"ExceptionGuideDeSurvie"
➥ reason:@"À titre d'exemple" userInfo:nil];
//3: méthode correcte
[NSException raise:@"ExceptionGuideDeSurvie"
➥ format:@"Montrer un exemple à nos lecteurs!"];
//4: autre manière correcte de procéder
// donne respectivement:
*** Terminating app due to uncaught exception
➥ of class 'NSCFString'
*** Terminating app due to uncaught exception
➥ of class 'NSCFNumber'
*** Terminating app due to uncaught exception
➥ 'ExceptionGuideDeSurvie', reason: 'À titre
➥ d'exemple'
*** Terminating app due to uncaught exception
➥ 'ExceptionGuideDeSurvie', reason: 'Montrer
➥ un exemple à nos lecteurs!'
```

Vous l'avez sans doute deviné, la classe `NSException` joue donc un rôle central dans la gestion des exceptions. Elle reste toutefois très simple. Chaque exception porte un nom (`name`), une explication pour l'utilisateur (`reason`) ainsi qu'un dictionnaire `userInfo`, s'il y a besoin de passer davantage d'informations au gestionnaire d'exceptions.

Gérer une exception

```
@try {
    //code susceptible de lever une exception
}
@catch (NSException* monException) {
    //traitement l'exception ici
}
@finally {
    //faire le nettoyage nécessaire
}
```

Objective-C emploie les directives `@catch()` et `@finally` là où Java et C# utilisent les mots-clés `catch` et `finally` respectivement.

Le fonctionnement est relativement simple à comprendre :

- La directive `@catch()` capture les exceptions du spécifié (dans notre exemple, toute exception de type `NSException` au sens large). Vous pouvez alors traiter l'exception directement ou *via* un gestionnaire dédié.

- La directive `@finally` est assez spéciale : le code qui s'y trouve est *toujours* exécuté, qu'une exception soit lancée ou non. Par exemple, même si vous insérez `return` dans la clause `@catch()` ou `@try`, `@finally` sera quand même exécuté. Il est important d'insister sur ce concept car cette clause est très importante en pratique : elle permet d'effectuer le nettoyage qui est nécessaire, qu'une exception soit levée ou non (par exemple, libérer les connexions vers une base de données ou des gestionnaires de fichiers, etc.).

```
@try {
    [instance
    ➥ faireQuelqueChoseQuiLeveUneException];
}
@catch (NSException* monException) {
    //traiter l'exception ici
    NSLog(@"Une exception est survenue: %@",
    ➥ monException);
    [self gereException:monException];
}
@finally {
    //faire le nettoyage nécessaire
    [instance fermerLesConnexions];
}
```

Attention

Si vous utilisez la gestion manuelle de la mémoire (parce que vous développez une application iPhone par exemple), il faut faire attention à la libération de la mémoire. Il n'y a malheureusement pas de règle générale et il faut traiter au cas par cas. Mais si vous appelez des méthodes qui peuvent lever des exceptions, libérez vos objets en envoyant le message `release` dans la clause `@finally`.

Enfin, la clause `@finally` est exécutée même si le gestionnaire d'exceptions lance une exception car il a été incapable de gérer le précédent problème.

Attention

Nous avons vu que la clause `@finally` était toujours exécutée. Toutefois, cela de veut pas dire qu'elle a été complétée avec succès : si une exception est levée au cours de son traitement, le flux sera interrompu et l'exécution restera partielle.

Pour finir cette section, voyons l'utilisation de macros au lieu de directives pour la gestion des exceptions.

Comme nous l'avons vu précédemment, Objective-C ne disposait pas des directives compilateur de gestion d'exceptions avant la version 10.3 de Mac OS X.

Il fallait alors avoir recours aux macros `NS_DURING`, `NS_HANDLER` et `NS_ENDHANDLER` qui sont toujours disponibles pour des raisons de rétrocompatibilité.

Leur utilisation est bien sûr à proscrire dès que vous n'avez pas à gérer ces systèmes antiques. Nous ne les mentionnons ici qu'à titre informatif et pour éviter tout surprise au cas où vous les rencontreriez en parcourant du code qui ne serait pas récent.

Enfin, sur les systèmes disposant des exceptions, ces macros sont automatiquement converties au moment de la compilation en leur directive correspondante : `@try`, `@catch()`. Il n'y a pas d'équivalent de la directive `@finally`. Les plus curieux pourront se tourner vers le fichier `NSException.h` pour voir comment cette conversion a été implémentée par Apple.

Gérer partiellement une exception

```
@try {
    [instance
faireQuelqueChoseQuiLeveUneException];
}
@catch (GuideDeSurvieException* monException) {
    //traiter l'exception de type
    ➥GuideDeSurvieException ici
```

```
    NSLog(@"Une exception de type
    ➥ GuideDeSurvieException est survenue: %@",
    ➥ monException);
    @throw; //relance implicitement monException
}
```

Il est parfois utile de ne gérer que partiellement une exception. Par exemple, vous gérer localement une exception pour éviter à l'application de garder un état instable, mais vous avez besoin de faire remonter le problème car il n'est pas résolu.

La directive `@throw` dans une clause `@catch()` permet de lancer l'exception courante, sans avoir besoin de la spécifier explicitement.

Info

Comme nous l'avons vu à la section "Implémenter la méthode finalize", la clause `@finally` est exécutée même si l'exception est relancée. Cela impacte la gestion de la mémoire en mode géré et il faut être particulièrement attentif si vous créez des bassins de libération automatiques (instances de `NSAutoreleasePool`). Nous vous renvoyons vers la documentation d'Apple pour de plus amples informations sur ce problème précis.

Capturer plusieurs types d'exceptions

```
@try {
    [instance faireQuelqueChoseQuiLeveUneException];
}
@catch (GuideDeSurvieException* monException) {
    //traiter l'exception de type
    //GuideDeSurvieException ici
    NSLog(@"Une exception de type
    ➥ GuideDeSurvieException est survenue: %@",
    ➥ monException);
}
@catch (NSException* monException) {
    //traiter l'exception de type NSException ici
    NSLog(@"Une exception de type NSException est
    ➥ survenue: %@", monException);
}
@finally {
    //faire le nettoyage nécessaire
    [instance fermerLesConnexions];
}
```

En général, il est utile de traiter chaque type d'exception différemment. Par exemple, si votre code transfère des fichiers depuis un serveur FTP vers votre disque dur local, il faudra gérer d'une autre manière le cas où la connexion au serveur est interrompue et le cas où le disque dur local est plein.

Vous devez capturer les exceptions dans l'ordre du plus spécifique vers le plus général. Ainsi, dans

notre exemple, `GuideDeSurvieException` dérivant de `NSException`, il faut le capturer en premier :

```
Une exception de type GuideDeSurvieException est
survenue: Impossible de faire quelque chose!
```

Si vous capturez les exceptions dans le mauvais ordre, votre code sera incorrect. En inversant l'ordre des deux clauses `@catch()` dans notre exemple précédent, nous obtiendrions :

```
Une exception de type NSException est survenue:
Impossible de faire quelque chose!
```

Fort heureusement, vous n'avez pas à connaître toute la hiérarchie de toutes les exceptions car, cerise sur le gâteau, le compilateur effectue une vérification et vous avertit s'il détecte un problème (voir Figure 5.2).

```
@try {
    [instance faireQuelqueChoseQuiLeveUneException];
}
@catch (NSException* monException) {
 ⚠ warning:   by earlier handler for 'struct NSException'
    //traiter l'exception de type NSException ici
    NSLog(@"Une exception de type NSException est survenue: %@", monException);
}
@catch (GuideDeSurvieException* monException) {
 ⚠ warning: exception of type 'struct GuideDeSurvieException' will be caught
    //traiter l'exception de type GuideDeSurvieException ici
    NSLog(@"Une exception de type GuideDeSurvieException est survenue: %@", monException);
}
```

Figure 5.2 : L'ordonnancement des clauses `@catch()` est important, mais heureusement, le compilateur est là pour vous aider.

Conclusion : l'ordonnancement des directives `@catch()` est très important.

Capturer toutes les exceptions

```
@catch (GuideDeSurvieException* monException) { …
}
@catch (NSException* monException) { … }
@catch (id pratiqueNonRecommandable) { … }
```

Nous avons vu au cours de la section précédente qu'il est possible de lancer n'importe quel objet et que la directive `@throw` n'est pas limitée aux instances du type `NSException`.

Un programme très défensif capturera donc non seulement les instances de `NSException`, mais également les autres classes. N'ayant pas le contrôle sur les bibliothèques qu'il utilise, il se peut donc que le code externe qu'il exécute ne se conforme pas à cette recommandation.

La manière de procéder consiste donc à toujours capturer les objets quel que soit leur type (avec `id` comme dans l'exemple précédent).

```
@try {
    [instance
faireQuelqueChoseQuiLeveUneException];
}
@catch (GuideDeSurvieException* monException) {
    //traiter l'exception de type
    //GuideDeSurvieException ici
    NSLog(@"Une exception de type
    ➥GuideDeSurvieException est survenue: %@",
    ➥monException);
```

```
}
@catch (NSException* monException) {
    //traiter l'exception de type NSException ici
    NSLog(@"Une exception de type NSException est
    ➥ survenue: %@", monException);
}
@catch (id pratiqueNonRecommandable) {
    //traiter ici la capture d'un objet ne
    //dérivant pas de NSException
    NSLog(@"Un objet du type %@ ne dérivant pas
    ➥ de NSException a été capturé",
    ➥ [pratiqueNonRecommandable class]);
    NSLog(@"Penser à envoyer un exemple de cet
    ➥ ouvrage à l'auteur du code fautif!");
}
@finally {
    //faire le nettoyage nécessaire
    [instance fermerLesConnexions];
}
```

Que deviennent les exceptions non capturées ?

```
NSSetUncaughtExceptionHandler(&
➥ CapteurGlobalDexceptions);
```

Il est possible de définir un gestionnaire d'exceptions "attrape-tout" pour intercepter toute exception non capturée dans une clause `@catch()` et pouvoir,

par exemple, enregistrer l'erreur dans un fichier d'historique ou procéder à des nettoyages avant d'être terminé.

L'exemple suivant montre comment vérifier si un tel gestionnaire a déjà été défini (avec la fonction `NSGetUncaughtExceptionHandler`), et dans le cas contraire, comment le faire (avec la fonction `NSSetUncaughtExceptionHandler`).

Une fois l'exception traitée par le gestionnaire par défaut, le programme est terminé.

```
void CapteurGlobalDexceptions(NSException*
➥ exception){
    NSLog(@"L'exception suivante n'a pas été
    ➥ capturée : %@(raison: %@)", [exception
    ➥ name], [exception reason]);
}
//[…]
if(NSGetUncaughtExceptionHandler()){
    NSLog(@"Un gestionnaire d'exceptions par
    ➥ défaut est déjà en place");
}
else {
    NSLog(@"Pas de gestionnaire d'exceptions
    ➥ par défaut. Mettons le nôtre en place");
    NSSetUncaughtExceptionHandler(&
    ➥ CapteurGlobalDexceptions);
}
//[…]
@try {
    [instance
faireQuelqueChoseQuiLeveUneException];
}
```

```
@catch (GuideDeSurvieException* monException) {
    //traiter l'exception de type
    //GuideDeSurvieException ici
    NSLog(@"Une exception de type
    ➥ GuideDeSurvieException est survenue: %@",
    ➥ monException);
    @throw;
}
@catch (NSException* monException) {
    //…
}
@catch (id pratiqueNonRecommandable) {
    //…
}
@finally {
    //…
}
Pas de gestionnaire d'exceptions par défaut.
Mettons le nôtre en place
Une exception de type GuideDeSurvieException est
➥ survenue : Impossible de faire quelque chose!
L'exception suivante n'a pas été capturée :
ExceptionGuideDeSurvie (raison: Impossible de
➥ faire quelque chose!)
```

Info

Veuillez noter que le gestionnaire par défaut est défini pour les applications Cocoa : c'est la méthode d'instance **`reportException:`** de **`NSApplication`** qui s'occupe de *logger* l'exception (*via* **`NSLog()`**) avant que l'application ne soit terminée.

Pour aller plus loin

Pour peu que vous soyez prêt à saisir quelques commandes dans le Terminal et à faire quelques calculs basiques en binaire, il est possible de modifier le fonctionnement de l'environnement d'exécution d'Objective-C sous Mac OS X de manière à ce que l'application ne soit pas terminée de manière brutale lorsqu'une exception n'est pas capturée, qu'une adresse mémoire est invalide, ou bien encore qu'un message est envoyé à un objet déjà libéré.

Il est même possible de faire en sorte que les exceptions capturées laissent un `trace` dans un fichier historique que l'on pourra consulter ultérieurement à des fins de diagnostics.

Pour découvrir les fonctionnalités avancées proposées par la bibliothèque `ExceptionHandling`, reportez-vous à la documentation d'Apple.

Gérer correctement les erreurs avec `NSError`

Les méthodes qui souhaitent transmettre des informations sur une erreur survenue en cours de traitement doivent suivre la convention suivante :

- Le dernier paramètre qu'ils acceptent est un pointeur de type `NSError` (passé par référence).
- Si une erreur survient la méthode retourne, suivant le cas, `NO` ou `nil` (et non pas un code d'erreur,

un objet spécial, etc.) afin que le test d'erreur soit trivial : `if (myString == nil)` dans notre exemple.

- Si la méthode appelante souhaite obtenir des informations en cas d'erreur, elle passe un pointeur de type `NSError` par référence (`&erreurInterne` dans notre exemple). Sinon, elle peut simplement passer `nil`. Elle vérifiera ensuite sur le pointeur passé n'est plus `nul`, c'est-à-dire que l'instanciation de `NSError` n'a pas échoué : `if(erreurInterne)` dans notre exemple.

```
NSError *erreurInterne;
NSString *myString = [[NSString alloc]
➥ initWithContentsOfURL:url encoding:
➥ NSUnicodeStringEncoding
➥ error:&erreurInterne];
    if (myString == nil) {
    //une erreur est survenue
    if(erreurInterne){
    //la classe NSError contenant des
    //informations supplémentaires
    //a été instanciée
    //gérer l'erreur et avorter
    }
}
else {
    //pas d'erreur, on peut continuer...
    //[...]
}
```

Créer et configurer une instance de NSError

```
error = [NSError
➥ errorWithDomain:NSCocoaErrorDomain code:-1
➥ userInfo:nil];
// ou :
error = [[NSError alloc]
➥ initWithDomain:NSCocoaErrorDomain code:-1
➥ userInfo:nil];
```

Comme d'habitude avec les classes Objective-C, il est possible d'instancier un nouvel objet avec la combinaison `alloc/init` ou avec la méthode utilitaire de la classe-objet. Notez que si vous utilisez `alloc/init`, vous devez libérer l'objet par la suite, car la gestion de sa mémoire vous appartient.

`NSError` dispose d'un domaine, servant à identifier son origine. Apple définit quatre différents domaines, `NSCocoaErrorDomain`, `NSPOSIXErrorDomain`, `NSOSStatus-ErrorDomain` et `NSMachErrorDomain` dans le fichier d'en-tête `NSError.h`. De plus, vous pouvez définir vos propres domaines (veuillez vous reporter à la documentation décrivant les conventions à suivre).

Le code d'erreur permet également d'encapsuler dans un objet `NSError` le code reçu depuis une fonction de bas niveau (typiquement une fonction C).

Enfin, le dictionnaire `userInfo` se comporte de la manière semblable à celui de `NSException` et contient

les informations supplémentaires sur l'erreur et la manière de le résoudre.

La convention est toutefois que `userInfo` doit contenir les éléments suivants :

- `NSLocalizedDescriptionKey`. Description de l'erreur survenue, par exemple : "Impossible de se connexion au serveur FTP".
- `NSLocalizedFailureReasonErrorKey`. Informations supplémentaires sur les raisons de l'échec, par exemple :"L'authentification a échoué".
- `NSLocalizedRecoverySuggestionErrorKey`. Proposition sur la manière de résoudre le problème, par exemple : "Souhaitez-vous saisir à nouveau votre identifiant et mot de passe ?".
- `NSLocalizedRecoveryOptionsErrorKey`. Un tableau de chaînes identifiant les boutons à apparaître sur le message d'erreur. Par exemple : "Réessayer", "Abandonner".
- `NSRecoveryAttempterErrorKey`. Un objet se conformant au protocole informel `NSErrorRecovery-Attempting` permettant d'essayer de résoudre le problème rencontré.
- `NSUnderlyingErrorKey`. Il est possible qu'une erreur soit remontée d'un système, et soit encapsulée dans une autre erreur afin de la rendre plus claire ou de fournir des informations supplémentaires.

  Attention toutefois : à la différence de `NSException`, il ne faut pas accéder directement aux éléments de `userInfo`. En effet, `NSError` étant également

destiné à afficher un message d'erreur à l'utilisateur, celui-ci peut être localisé.Vous devez donc obtenir les informations en utilisant les méthodes d'instances, respectivement :

- localizedDescription
- localizedFailureReason
- localizedRecoverySuggestion
- localizedRecoveryOptions

- `recoveryAttempter` (c'est un objet et non une chaîne, mais un accesseur existe toutefois).
- Il n'y a pas d'accesseur pour l'erreur sous-jacente, vous l'accéderez donc directement *via* la clé `NSUnderlyingErrorKey`.

Retourner une erreur

```
NSDictionary* infoUtilisateur = [NSDictionary
➥dictionaryWithObjectsAndKeys: @"Une erreur est
➥ survenue en faisant quelque chose de très
➥ compliquée.", NSLocalizedDescriptionKey, @"La
➥ fonction très compliquée a échouée",
➥ NSLocalizedFailureReasonErrorKey, nil];
*error = [NSError
➥ errorWithDomain:NSPOSIXErrorDomain
➥ code:errorCode userInfo:infoUtilisateur];
```

L'extrait ci-dessus montre comment `NSError` est instancié avec un dictionnaire contenant les détails sur l'erreur à transmettre.

Dans notre premier exemple ci-dessous, nous montrons comment encapsuler un code d'erreur d'une fonction C dans un objet `NSError` afin de le rendre utilisable et de l'enrichir d'informations pour l'utilisateur et pour Cocoa.

```
extern int function_posix_tres_compliquee(int*);
//exemple 1: création d'un objet NSError pour
➥ encapsuler un
//code erreur
- (BOOL)faireQuelqueChose:(NSInteger*)valeur
 ➥ erreur:(NSError**)error {
    int errorCode = function_posix_tres
    ➥_compliquee(valeur);
    if(errorCode == 0){ // pas d'erreur
        return YES;
    }
    else {
        if(error){// une erreur est survenue:
                  // errorCode != 0
            NSDictionary* infoUtilisateur =
            ➥ [NSDictionary
            ➥ dictionaryWithObjectsAndKeys:
            ➥@"Une erreur
            ➥ est survenue en faisant quelque
            ➥ chose de très compliqué.",
            ➥ NSLocalizedDescriptionKey, @"La
           ➥ fonction très compliquée a échouée",
            ➥ NSLocalizedFailureReasonErrorKey,
            ➥ nil];
            ➥*error = [NSError errorWithDomain:
```

```
                ➥ NSPOSIXErrorDomain code:errorCode
                ➥ userInfo:infoUtilisateur];
            }
            return NO;
        }
    }
```

Dans ce second cas, nous encapsulons une erreur qui a été retournée par une autre classe afin de l'enrichir d'informations et également de faire en sorte de propager une erreur de plus haut niveau que l'erreur originale (par exemple, un utilisateur de `telechargerPage:erreur:` n'a pas besoin de savoir le détail de l'erreur reçue par `NSString`).

À noter ici que nous utilisons `NSUnderlyingErrorKey` afin d'inclure l'erreur originale dans notre nouvelle erreur et de ne pas perdre d'information en cours de route.

```
//exemple 2: création d'un objet NSError pour
➥ encapsuler un
//autre NSError plus basique
- (BOOL)telechargerPage:(NSURL*)url
➥ erreur:(NSError**)error
{
    NSError *erreurInterne;
    NSString *myString = [[NSString alloc]
    ➥ initWithContentsOfURL:url encoding:
    ➥ NSUnicodeStringEncoding
    ➥ error:&erreurInterne];
```

```
        if (myString == nil && erreurInterne) {//une
        ➥ erreur est survenue
            NSDictionary* infoUtilisateur =
            ➥ [NSDictionary
            ➥ dictionaryWithObjectsAndKeys:
            ➥ erreurInterne, NSUnderlyingErrorKey,
            ➥@"Une erreur est survenue lors du
            ➥ téléchargement",
            ➥ NSLocalizedDescriptionKey, nil];
           *error = [NSError
           ➥ errorWithDomain:NSPOSIXErrorDomain
           ➥ code:[erreurInterne code]
           ➥ userInfo:infoUtilisateur];
          return NO;
       }
       else {
           return YES;
       }
   }
```

Important

Afin que la méthode appelée puisse instancier l'objet que vous lui passez et la retourner, elle doit recevoir par référence un pointeur vers le pointeur local, d'où `(NSError**)error` et `error:&erreurInterne` dans l'exemple ci-dessus.

Gérer les erreurs

```
NSError* erreur //instanciée…
//1 - Demander à NSApp d'afficher l'erreur
//directement :
[NSApp presentError:erreur];
//2 - Ou créer un objet NSAlert et l'afficher :
[[NSAlert alertWithError:erreur] runModal];
```

La classe `NSAlert` d'AppKit permet d'afficher une fenêtre d'erreur standard contenant les informations de l'instance de `NSError` qui lui a été transmise *via* les méthodes `beginSheetModalForWindow:modalDelegate:didEndSelector:contextInfo:` et `alertWithError:`. La classe `UIAlertView` est l'équivalent UIKit de `NSAlert` et permet d'afficher le message d'erreur sur iPhone.

Toutefois, il est en général plus utile de connaître l'existence de la méthode `presentError:` de la classe `NSResponder` dont héritent des classes aussi importantes que `NSApplication`, `NSWindow`, `NSView` et `NSWindowController`.

Il est ainsi possible de demander à votre application de présenter directement le message d'erreur en envoyant le message `presentError:` à l' instance de `NSApplication`.

```
NSArray* boutons = [[NSArray alloc]
➥ initWithObjects:@"Oui", @"Non", @"Peut-être",
➥ nil];
```

```
NSDictionary* infoUtilisateur = [NSDictionary
➥ dictionaryWithObjectsAndKeys:
       @"Une erreur est survenue lors de la
       ➥ connexion au serveur.",
       ➥ NSLocalizedDescriptionKey,
       @"Le serveur FTP est indisponible ou la
       ➥ connexion Internet est coupée",
       ➥ NSLocalizedFailureReasonErrorKey,
       @"Vérifiez votre connexion et essayez de
       ➥ nouveau.",
       ➥ NSLocalizedRecoverySuggestionErrorKey,
boutons,NSLocalizedRecoveryOptionsErrorKey, nil];
//1 - Demander à NSApp d'afficher l'erreur
//directement :
[NSApp presentError:[NSError
➥ errorWithDomain:NSCocoaErrorDomain code:-1
➥ userInfo:infoUtilisateur]];
//2 - Ou créer un objet NSAlert et l'afficher:
[[NSAlert alertWithError:[NSError errorWithDomain:
➥ NSCocoaErrorDomain code:-1
➥ userInfo:infoUtilisateur]] runModal];
```

Info

Par soucis de simplicité et de lisibilité, nous n'avons pas utilisé les fonctions de localisations ici, mais pour avoir un code de meilleure qualité, vous devez insérer les éléments dans le dictionnaire avec la macro **`NSLocalizedString()`**. Pour plus d'informations sur les fonctions de localisation de Cocoa, veuillez-vous reporter à la documentation d'Apple.

Enfin, pour utiliser au mieux les erreurs, il faut apprendre à employer la chaîne des répondeurs d'erreurs (voir encadré ci-dessous).

Pour aller plus loin

Chaîne des répondeurs d'erreurs.

Nous avons vu comment présenter de manière simpliste une erreur à l'utilisateur. Toutefois, afin d'utiliser au mieux les possibilités offertes par l'intégration de `NSError` avec Cocoa sur Mac OS X (actuellement indisponible sur la plateforme iPhone), il est nécessaire de bien comprendre la *chaîne des répondeurs d'erreurs* (*error-responder chain*) et la manière de faire remonter une erreur dans la chaîne tout en l'enrichissant d'informations. Nous vous incitons vivement à vous reporter à la documentation d'Apple sur ces points.

Recouvrer une erreur avec un objet implémentant `NSRecoveryAttempting`.

Il est possible de passer au dictionnaire de NSError un objet implémentant le protocole informel `NSRecoveryAttempting` pour la clé `NSRecoveryAttempterErrorKey` afin de fournir une solution au problème rencontré. Toutefois, cette fonctionnalité avancée est relativement peu utilisée puisqu'il s'agit en général de suggérer à l'utilisateur d'effectuer des vérifications manuelles avant de faire une nouvelle tentative.

Nous vous invitons bien sûr à consulter la documentation d'Apple si vous êtes intéressé par les possibilités offertes par ce mécanisme.

NSException

À la différence de Python où la levée d'exception est le mode recommandé de propagation des erreurs, et de Java ou C# où la levée d'exception est coûteuse, mais toutefois courante, Apple considère la gestion d'exception comme un outil d'aide au développement et servant principalement à découvrir et supprimer les erreurs de programmations à proprement parler.

Cela signifie que toute erreur prévisible provenant du milieu extérieur au code doit être gérée *via* les erreurs (`NSError`). Ceci est notamment le cas d'un serveur indisponible pour une application de transfert de fichiers ou bien encore un mot de passe invalide lors d'une authentification. Il ne s'agit pas d'erreurs de programmation, mais d'erreurs attendues : il faudra utiliser `NSError`.

Un bon moyen de savoir si une exception doit être levée ou une erreur reportée est de se demander l'application doit être terminée ou non. Si l'application peut rétablir la situation, par exemple *via* une action de l'utilisateur, cela signifie qu'une erreur doit être reportée et l'exception évitée.

Info

La classe `NSError` est apparue avec Mac OS X 10.3. Il est donc possible, si vous utilisez une bibliothèque relativement ancienne, que cette dernière lève des

exceptions là où il faudrait reporter une erreur. Dans ce cas, vous ne devez pas propager l'exception. Vous devez la capturer et reporter une erreur *via* une nouvelle instance de la classe `NSError` en utilisant les informations contenues dans l'objet `NSException`.

L'utilisation du mécanisme d'exception introduit, malgré toutes les optimisations apportées au fils des ans, une perte en performance qualifiée d'importante par Apple. Ainsi, par exemple, en mode 32 bits, les clauses `@try` sont extrêmement coûteuses, même si le coût de `@throw` est modéré. En mode 64 bits, les clauses `@try` ont connu un changement important qui rend leur utilisation gratuite, mais `@throw` est en conséquence devenu bien plus coûteuse qu'en mode 32 bits.

Même s'il ne paraît pas évident à la première lecture, le mode 64 bits apporte donc une amélioration importante par rapport au mode 32 bits : en 32 bits, vous êtes pénalisé dès que vous entrez `@try` même si aucune exception n'est levée. En mode 64 bits, vous pouvez utiliser `@try` autant que vous le souhaitez, tant que vous ne lancez pas d'exception, il n'y aura pas d'impact en termes de performance.

Cela signifie que, si vous suivez les recommandations d'Apple, votre code sera bien plus rapide en mode 64 bits qu'en mode 32 bits car vous êtes prêt à capturer les exceptions. Mais il ne devrait y avoir aucune levée d'exception sauf en

cas d'erreur grave *et* exceptionnelle comme, par exemple, l'impossibilité d'allouer de la mémoire (ce qui, en théorie est impossible sur un système utilisant de la mémoire virtuelle, à moins que la mémoire et le disque dur soient tous deux saturés).

NSError

Apple a introduit la classe `NSError` afin de supprimer les codes cryptiques qui étaient utilisés pour reporter les erreurs (qui n'a pas déjà vu une application planter en raison de l'erreur `-894109` ?) mais également pour éviter l'utilisation des exceptions qui sont très coûteuses en termes de performance.

L'utilisation des erreurs est donc préconisée dans tous les cas où le problème provient de paramètres externes au code source et qui sont en général clairement prévisible : une application FTP qui n'arrive pas à se connecter à un serveur car la connexion interne est coupée ne devra pas lever d'exception pour communiquer le problème car c'était un cas tout à fait prévisible. De même, si l'authentification d'un utilisateur à un serveur FTP échoue, il faudra reporter une erreur et non pas lancer une exception.

En conséquence, `NSError` devra être utilisée dans la quasi-totalité des cas où une erreur doit être communiquée à l'utilisateur, les *exceptions* étant à réserver aux cas *exceptionnels* et aux erreurs de programmation.

Créer une assertion

```
NSAssert(valeur < 255 && valeur > 0, @"valeur
n'était pas compris dans la zone déterminée");
```

Une assertion est un outil d'aide au débogage de code et permet au développeur de se rendre compte lorsqu'un objet ou une valeur ne correspond pas à ses attentes. Ils ne sont donc pas propres à Objective-C et existent en C, C++, Java, C#, etc.

Vous pouvez utiliser les fonctions `NSAssert` et ses dérivées (`NSAssert1` à `NSAssert5`) pour détecter des valeurs incorrectes ou invalides dans votre code en amont de la gestion d'erreur : les assertions ne vous dispensent pas de faire de la programmation défensive, bien au contraire.

Lorsqu'une assertion échoue (c'est-à-dire que la condition que vous avez définie est fausse), `NSAssert` imprime une erreur sur la console et lève une exception. Cela permet de repérer le problème immédiatement et de le corriger, en évitant que les mécanismes de défense que vous avez mis en place ne déplace le problème à un niveau supérieur.

Une exception est levée lorsqu'une assertion échoue. Or, comme nous l'avons précédemment, les exceptions en Objective-C ne doivent pas être générée en cours d'exécution, sauf lorsque l'application doit être terminé. Cela confirme leur rôle d'outil d'aide au développement, mais qu'ils ne doivent pas être livré avec le code en production.

Supprimer les assertions pour le code en production

```
#define NS_BLOCK_ASSERTIONS
```

Les assertions ne seraient pas utilisables en pratique s'il fallait parcourir tout le code et les supprimer une par une avant de publier une nouvelle version de votre application.

Heureusement, le compilateur vous aide dans cette tâche : il ignorera les assertions dans chaque fichier où la variable préprocesseur `NS_BLOCK_ASSERTIONS` est définie.

Vous supprimez toutes les assertions en définissant la macro préprocesseur directement *via* l'inspecteur de projet (voir Figure 5.3).

Enfin, vous pouvez décider de supprimer toutes les assertions lorsque vous êtes en mode release avec les directives suivantes :

```
#ifndef DEBUG
#define NS_BLOCK_ASSERTIONS
#endif
```

Les développeurs les plus avancés pourront tirer parti de la possibilité de créer des fichiers de configuration pour Xcode (fichiers `.xcconfig`) afin d'y définir `NS_BLOCK_ASSERTIONS` et ainsi supprimer automatiquement toutes les assertions d'une certaine *build*.

Figure 5.3 : L'onglet Build de l'inspecteur de projet d'Xcode permet de modifier les macros du préprocesseur. Vous noterez bien les mot-clés utilisés dans le champ de recherche.

Consigner des messages informatifs ou des erreurs

```
NSLog(@"Notification %@ envoyé par %@ et info %@",
➥ notif.name, notif.object, notif.userInfo);
```

La fameuse `NSLog()`, sans doute la plus utilisée des fonctions de la *Foundation framework*, permet

d'imprimer une chaîne de caractères sur la sortie erreur standard (`stderr`). Ces messages sont également enregistrés *via* le système d'historique standard du système d'exploitation (par exemple, visible à la Figure 5.4 dans l'application Console de Mac OS X).



Figure 5.4: L'application Console de Mac OS X permet de voir les messages enregistrés *via* la méthode `NSLog()`.

Configurer son projet pour les tests unitaires

Apple intègre, depuis Xcode 2.1, la bibliothèque open-source de tests unitaires *OCUnit*, développée par la société suisse SenTe.

Test unitaire

Étant un grand supporter des méthodologies de développement dites agiles (tels que *XP* et *Scrum*), et pratiquant de TDD (*Test Driven Developement*), il m'est quasiment impossible de concevoir du code

sans tests unitaires. Si vous êtes intéressé par découvrir ces méthodologies et améliorer votre productivité par un facteur de l'ordre 200 à 300 %, rendez-vous sur InfoQ.com où vous aurez accès à de nombreuses présentations par les créateurs des différentes méthodologies Agiles.

Si vous n'avez jamais encore utilisé les tests unitaires, nous vous conseillons vivement de les découvrir et de les intégrer à votre code et profiter ainsi de leurs très nombreux avantages, parmi lesquels nous pouvons citer :

- **Aider à comprendre le besoin en écrivant le test avant même d'avoir écrit le corps de la méthode.**
- **Capturer toute régression dans le code et réduire énormément le risque d'introduction de nouveau bogues.**
- **Documenter le code en fournissant des exemples concrets d'utilisation.**

Nous allons configurer le projet Xcode de manière à ce qu'à chaque fois que le code est compilé, les tests unitaires soient exécutés. Ce n'est malheureusement pas une tâche triviale, c'est pourquoi nous allons suivre minutieusement la documentation d'Apple.

1. Sélectionnez Project > New Target…
2. Sélectionnez Cocoa, puis Unit Test Bundle.
3. Donnez un nom à la nouvelle cible, puis cliquez sur OK.

4. L'inspecteur devrait alors s'ouvrir automatiquement. Si ce n'est pas la cas, ouvrez-le.
5. Sous l'onglet General de l'inspecteur, cliquez sur le bouton + qui se trouve en dessous de la zone Direct Dependencies (voir Figure 5.5).



Figure 5.5 : L'onglet General de l'inspecteur de projet d'Xcode permet de rendre le projet contenant les tests unitaires dépendant du projet de l'application à compiler.

6. Sélectionnez alors la cible qui correspond à votre projet principal (par exemple, si vous développez une application et plusieurs bibliothèques, sélectionnez l'application) et cliquez sur "Add Target".
7. Si vous développez une bibliothèque, vous n'avez plus rien à faire. En revanche, si c'est une application que vous développez, cliquez sur l'onglet "Build" et recherchez "Bundle Loader" (dans le champ recherche dédié).
8. Saisissez `$(CONFIGURATION_BUILD_DIR)/GuideDeSurvie.app/Contents/GuideDeSurvie` dans le champ "Value" associé (en replaçant `GuideDeSurvie` par votre application bien sûr).
9. Toujours sous l'onglet "Build", recherchez "Unit Testing" et associez la même valeur au champ "Test Host" ; dans notre cas, c'est la valeur `$(CONFIGURATION_BUILD_DIR)/GuideDeSurvie.app/Contents/GuideDeSurvie` que nous lui associons (voir Figure 5.6).
10. Enfin, sélectionnez votre nouvelle cible afin de la rendre active. À partir de cet instant, à chaque fois que vous compilerez le projet, la cible principale sera compilée (car nous avons indiqué que la cible de tests unitaires était dépendante de celle-ci), ensuite les tests seront exécutés.

Voilà ! Vous avez créé une cible pour vos tests unitaires et ces tests seront exécutés automatiquement à chaque fois que vous compilerez votre projet !

Figure 5.6 : L'onglet Build de l'inspecteur de projet d'Xcode permet de modifier les réglages des tests unitaires. Vous noterez bien les mot-clés utilisés dans le champ de recherche.

Créer une classe de tests

1. Sélectionnez File > New File.
2. Sélectionnez Cocoa, puis "Objective-C test case file", donnez un nom et laissez la case "Also create NomDeFichier.h" cochée.
3. Changez la cible par défaut pour que le fichier soit inclus uniquement dans la cible de tests unitaires que nous avons créée à la section précédente.

Nous écrivons alors notre première fonction de test unitaire, la plus basique qui soit, afin de nous assurer que nous avons bien configuré le projet :

```
@implementation TestsUnitaires
- (void) test1{
    NSLog(@"test1");
    STAssertNotNil(nil, @"assert not nil");
}
@end
```

Bien évidemment, ce test est voué à échouer dès la prochaine compilation du projet ! Grâce à cet échec, nous pouvons confirmer qu'Xcode exécute bien nos tests (voir Figure 5.7).

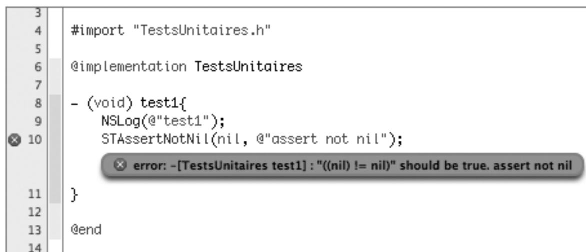


Figure 5.7 : Le compilation échoue lorsque l'un des tests unitaires échoue.

Écrire des tests unitaires

```
- (void) testInit{
    ClasseGuideDeSurvie * instance =
    ➥ [[ClasseGuideDeSurvie alloc] init];
    STAssertNotNil(instance, @"assert not nil");
}

- (void) testMaChaine{
    ClasseGuideDeSurvie * instance =
    ➥ [[ClasseGuideDeSurvie alloc] init];
    [instance setMaChaine:@"testChaine"];
    STAssertEquals(instance.maChaine, @"testChaine",
    ➥@"maChaine n'a pas retournée la valeur
    ➥ attendue.");
}
```

Comme toutes les plateformes de tests unitaires (par exemple, *JUnit*, *NUnit*, *MbUnit*, etc.), les méthodes de tests unitaires en Objective-C doivent se conformer aux règles suivantes :

- Retourner `void`.
- Ne pas prendre de paramètres.
- Être préfixées par `test` (ceci est une convention de nommage).

Elles seront donc toutes de la forme du code ci-dessus.

De même, les méthodes `setUp` et `tearDown` fonctionnent de manière semblable à celles portant le même nom dans les bibliothèques *xUnit*.

La méthode `setUp` est exécutée une seule fois avant que les tests de la classe ne soit lancés, tandis que la méthode `tearDown` est exécutée une fois que tous les tests ont été exécutés. Ces deux méthodes permettent donc de préparer et de terminer les tests (par exemple, `setUp` peut récupérer des données d'une base de données une seule fois, et préparer les objets).

Enfin, pour exécuter les tests à proprement parler, vous disposez de nombreuses macros, dont les plus courantes sont :

- `STAssertNotNil(a1, description, ...)`. L'expression `a1` ne doit pas être égale à `nil`.
- `STAssertTrue(expression, description, ...)`. L'expression doit être évaluée à `YES`.
- `STAssertFalse(expression, description, ...)`. L'expression doit être évaluée à `NO`.
- `STAssertEqualObjects(a1, a2, description, ...)`. L'expression `[a1 isEqualTo:a2]` doit être évaluée à `YES`.
- `STAssertEquals(a1, a2, description, ...)`. L'expression `a1 == a2` doit être évaluée à `YES`. À reserver pour les *structs* et valeurs intégrales, etc.

- `STAssertThrows(expression, description, ...)`. Le test doit capturer une exception pour passer. Si vous utilisez ce test, demandez-vous si vous ne devriez pas utiliser `NSError` au lieu de `NSException` !

Le code source de SenTestingKit

Vous pouvez découvrir le code source de la bibliothèque de tests unitaires d'Objective-C en ligne, ou bien sur Mac OS X : **`/System/Library/Frameworks/SenTestingKit.framework`**.

Le fichier **`SenTestCase_Macros.h`** contient l'ensemble des macros définies pour les tests. Vous verrez qu'ils ne sont pas si compliqués à implémenter !

Annexes

A

Objective-C++

Mélanger du code Objective-C et C++ et créer Objective-C++

```
#import <Foundation/Foundation.h>
class Bonjour {
private:
    NSString* message_salutation;
public:
    Bonjour() {
        message_salutation = @"Bonjour!";
    }

    Bonjour(const char* initial_message_
    ➥salutation) {
        message_salutation = [[NSString alloc]
        ➥ initWithUTF8String:initial_message_
        ➥ salutation];
    }
```

```
    void afficher_saluation() {
        printf("%s\n", [message_salutation
        ➥ UTF8String]);
    }
};

@interface Salutation : NSObject {
@private
    Bonjour * bonjour;
}
- (id)init;
- (void)dealloc;
- (void)saluer;
- (void)saluer:(Bonjour*)bonjourIntance;
@end
```

L'exemple précédent, adapté de la documentation d'Apple, permet de parfaitement illustrer l'intégration de C++ dans Objective-C pour donner naissance à Objective-C++.

Il est désormais possible de définir des classes C++ disposants de membres Objective-C et réciproquement.

La possibilité d'écrire du code C++ dans un fichier source Objective-C, et ainsi de pouvoir mélanger les deux langages, repose essentiellement sur la capacité offerte par Objective-C++ d'utiliser les pointeurs de manière transparente dans les deux langages et donc en conséquence directe, d'avoir des membres définis en C++ dans les classes Objective-C et réciproquement. C'est la principale règle à connaître pour pouvoir exploiter

Objective-C++ car c'est également la seule chose qu'il vous sera réellement permis de faire. Les possibilités offertes par Objective-C++ restent toutefois limitées (voir section "Limites d'Objective-C++").

Enfin, veuillez noter qu'un fichier source Objective-C++ doit porter l'extension `.mm` afin d'être reconnu par le compilateur.

Utiliser Objective-C++

Commençons par un peu d'histoire. Objective-C++ est apparu en 2002, avec la sortie de Mac OS X 10.1, et a pour principale raison d'être l'existence de nombreuses bibliothèques C++, jusque là inaccessibles des développeurs Objective-C.

Il faut se rappeler que lorsqu'Apple a enfin publié la version 10.0 de Mac OS X, la santé d'Apple et de la plateforme Mac étaient plus que menacées. Il était par ailleurs difficile de convaincre les développeurs d'adopter un nouveau langage, une nouvelle plateforme et en plus abandonner l'ensemble de leurs bibliothèques, souvent écrites en C++.

L'arrivée d'Objective-C++ avec Mac OS X 10.1 fut donc perçue comme une très bonne nouvelle par les développeurs, qui l'accueillirent avec beaucoup d'engouement.

Le principal cas d'utilisation d'Objective-C++ est donc l'intégration de bibliothèques C++ existantes.

Un autre cas corollaire, que l'on peut également citer, est celui où les performances d'Objective-C sont insuffisantes pour certains cas très particuliers. Une technique d'optimisation très poussée consiste alors à réécrire les algorithmes et certaines classes en C++ (imaginons par exemple que vous avez créé un nouvel algorithme révolutionnaire de compression d'images). Toutefois, avant d'arriver à ce cas extrême, il vaut mieux se mieux se remettre en cause plutôt que de soupçonner le langage.

Limites d'Objective-C++

Les possibilités offertes par Objective-C++ étant très restreintes, les limites sont, en conséquence, très nombreuses. Nous allons lister les principales restrictions et nous laisserons le lecteur intéressé se reporter à la documentation d'Apple pour les autres.

Le fonctionnement général des classes n'étant pas modifié, il faut considérer que le code s'exécute dans deux environnements totalement distincts et séparés. Ces environnements coexistent mais en gardant leurs propres règles et principes de fonctionnement.

Les principales actions que vous ne pouvez pas faire sont :

- Envoyer un message à un objet C++.
- Ajouter de constructeur ou destructeur à une classe Objective-C.

- Dériver une classe Objective-C d'une classe C++ et réciproquement.
- Utiliser `this` au lieu de `self` dans une méthode Objective-C et réciproquement.
- Capturer une exception C++ avec une directive `@catch` ou capturer une exception Objective-C avec une clause `catch` C++.

Enfin, au cas où vous ne trouverez pas ces restrictions suffisantes, il en existe une autre encore bien plus limitante : il est impossible d'utiliser une instance d'une classe C++ comme membre d'une classe Objective-C si, et seulement si, toutes les fonctions membres de la classe C++ sont non-virtuelles.

B

Ressources utiles

Le site Apple pour les développeurs

http://developer.apple.com/

Étant donné que vous allez développer pour l'une des plateformes Apple (Mac ou iPhone), c'est bien évidemment sur le site d'Apple consacré aux développeurs que vous vous rendrez le plus souvent.

Beaucoup de vétérans du Mac ont tendance à sous-estimer le site d'Apple car au cours des premières années de Mac OS X, la documentation avait tendance à être très succincte, voire manquante, et c'était la principale critique que recevait Apple. Depuis, la tendance a été largement renversée.

http://developer.apple.com/technology/xcode.html

Vous pouvez télécharger la dernière version d'Xcode pour iPhone et Mac gratuitement depuis ce lien.

Vous devez, en règle générale, vous connecter avec votre identifiant ADC (*Apple Developer Connection*) avant de pouvoir télécharger les outils et exemples. il existe différents type de compte, dont le compte ADC Online, qui est gratuit.

http://developer.apple.com/mac/

Le *Mac Developer Center* dispose de toute la documentation nécessaire à la programmation sur Mac (notamment Objective-C et Cocoa en ce qui concerne), mais également de nombreux exemples de code (*sample code*).

http://developer.apple.com/iphone/

L'*iPhone Developer Center* est l'équivalent du *Mac Developer Center*, mais pour les développeurs iPhone. Vous y trouverez donc un contenu similaire : documentation Objective-C et CocoaTouch, exemples de code, etc.

http://developer.apple.com/adconitunes

Ce site permet d'accéder à des vidéos d'introduction et d'entraînement que vous pouvez télécharger depuis iTunes et regarder sur votre Mac ou dans les transports en commun sur votre iPod ou iPhone.

http://developer.apple.com/iphone/program/

Afin de pouvoir déployer des applications maisons sur votre iPhone ou iPod Touch, vous devez disposer d'une licence développeur coûtant 99 dollars. Cette licence permet d'utiliser votre iPod ou iPhone en mode développement et vous autorise non seulement à publier vos logiciels sur la boutique en ligne iTunes, mais également à les installer sur une centaine d'appareils sans passer par iTunes *via* le mode particulier, appelé *Ad Hoc*. Vous devez toutefois enregistrer vos appareils sur votre compte iPhone Developer en suivant les explications qu'Apple vous fournira.

http://devworld.apple.com/wwdc/

La *WorldWide Developer Conference* est la conférence annuelle organisée par Apple (depuis quelques années au Moscone Center à San Francisco). Les développeurs du monde entier s'y retrouvent pour assister aux présentations données par les experts d'Apple, découvrir les nouvelles fonctionnalités d'Objective-C ou de Cocoa et assister à la *messe* donnée par Steve Jobs. Si vous souhaitez devenir un professionnel de la programmation Apple, vous rendre à cette conférence est recommandée (à condition bien sûr d'avoir le temps et les moyens).

http://developer.apple.com/products/videos.html

Toutes les présentations de la WWDC sont disponibles depuis ce site. Elles sont accessibles gratuitement aux personnes qui se sont rendues à la conférence.

Documentations Apple recommandées

Ce livre vous donne les bases d'Objective-C (mais aussi dans certains cas, de Cocoa) et les cas d'utilisation les plus courants des directives, concepts et classes. Mais afin d'améliorer votre compréhension de la plateforme, de maîtriser au mieux les concepts et bibliothèques, nous vous recommandons de lire au moins les documents suivants d'Apple (en anglais). Ces documents sont installés avec Xcode et disponibles gratuitement depuis le site d'Apple[1] :

- *The Objective-C 2.0 Programming Language*
- *Coding Guidelines for Cocoa*
- *Memory Management Programming Guide for Cocoa*
- *Garbage Collection Programming Guide*
- *Cocoa Event Handling Guide*
- *Notification Programming Topics for Cocoa*

1. Vous trouverez tous ces documents depuis : **http://developer.apple.com/mac/library/navigation/**.

- *Exception Programming Topics for Cocoa*
- *Error Handling Programming Guide For Cocoa*
- *Xcode Unit Testing Guide*

Voici une petite liste de documents qui traitent de sujets non couverts dans cet ouvrage, mais qui nécessitent toutefois d'être maîtrisés avant de pouvoir écrire des applications tirant au mieux parti des possibilités offertes par Cocoa :

- *Cocoa Fundamentals Guide* (requis pour bien comprendre les concepts définis et utilisés par Cocoa)
- *InterfaceBuilder User Guide* (requis pour l'utilisation d'InterfaceBuilder)
- *Key-Value Coding Programming Guide* (requis pour l'utilisation des bindings)
- *Key-Value Observing Programming Guide* (requis pour l'utilisation des bindings)
- *Cocoa Bindings Programming Topics* (requis pour l'utilisation des bindings)
- *Model Object Implementation Guide* (requis pour l'utilisation de CoreData)
- *Predicate Programming Guide* (requis pour l'utilisation de CoreData)

Les listes précédentes ne sont bien sûr pas exhaustives, et il y a de nombreux autres documents que vous serez amené à lire suivant les classes que vous utiliserez et les besoins de vos applications.

Sites intéressants

Sites francophones

Google.fr

http://www.google.fr

Je suis surpris par le nombre de développeurs qui n'ont toujours pas le réflexe de rechercher de l'aide sur Google dès que le moindre problème survient. Et bien sûr, ne limitez pas vos recherches aux sites francophone, car les ressources sont très rares : les recherches en anglais vous fourniront bien plus de résultats.

Page personnelle de Pierre Chatelier

http://pierre.chachatelier.fr/programmation/objective-c.php

Pierre Chatelier a écrit *De C++ à Objective-C*, un livre électronique qu'il met gratuitement à disposition de la communauté, et, qui comme son nom l'indique, permet aux développeurs C++ d'apprendre rapidement Objective-C.

Cocoa.fr

http://www.cocoa.fr/

Site français, présenté sous la forme d'un blog, et traitant de différents sujets relatifs à la programmation Cocoa.

Project:Omega

http://www.projectomega.org/

Site communautaire que j'ai cofondé il y a de nombreuses années avec mon ami Thierry. Le site contient de nombreux articles en Français sur la programmation Cocoa. Le site est toutefois rarement mis à jour par faute de temps et cherche de nouveaux contributeurs.

Sites anglophones

Cocoa Dev Central

http://cocoadevcentral.com/

Site consacré à la programmation Cocoa et proposant une énorme quantité d'exemples et de documents.

Mac Developer Network (MDN)

http://www.mac-developer-network.com/

Site communautaire proposant de nombreux articles, mais surtout et également un PodCast qui vous permettra d'approfondir Cocoa chez vous ou sur la route.

De plus, les éditeurs de MDN surveillent toute la blogosphère des développeurs Mac et publient sous forme de flux RSS leur sélection quotidienne des meilleurs articles. Ce flux était jusqu'à récemment proposé en échange d'une modique participation, mais il est désormais gratuit.

InfoQ

www.infoq.com

InfoQ est un site communautaire proposant de nombreux articles, présentations, interviews, sur de nombreux langages de programmation (Java, .Net, Ruby, pour n'en citer que quelques uns), mais malheureusement pas Objective-C. La raison laquelle ce site apparaît ici est qu'il regorge d'excellentes présentations sur les méthodes Agile, que je recommande vivement de regarder et de suivre *via* leur flux RSS.

CocoaDev

http://www.cocoadev.com/

CocoaDev est un Wiki contenant de très nombreux articles sur Objective-C et Cocoa et disposant d'une grande communauté d'utilisateurs, prêt à aider (parfois en échange d'une modique somme) les développeurs, par exemple, pour la traduction de logiciels dans divers langues.

Wil Shipley Blog

http://wilshipley.com/blog/

Wil Shipley est le fondateur de Delicious Monster, dont le logiciel Delicious Library ont bien connu des utilisateurs de Mac et a gagné de très nombreux prix depuis 2004. Il traite de nombreux sujets relatifs à Cocoa sur son blog, qui est bien évidemment vivement recommandé !

Cocoa With Love

http://cocoawithlove.com/

Blog d'un développeur australien qui travaille à plein temps sur Cocoa. Il fait partie de ma liste de flux et je le recommande chaudement.

TheoCacao

http://theocacao.com/

Blog d'un développeur américain qui est également le propriétaire de *Cocoa Dev Central*. Scott Stevenson est également en train d'écrire un livre en anglais sur Objective-C et Cocoa.

CocoaLab

http://www.cocoalab.com/

Ce site n'est plus mis à jour depuis 2007, mais propose toutefois *BecomeAnXcoder*, un livre électronique gratuit, mis à jour pour Mac OS X 10.5 sur Xcode que vous pouvez télécharger depuis le lien suivant : **http://www.cocoalab.com/?q= BecomeAnXcoder-Français**.

Google Objective-C Coding Style

http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml

Ce document vient s'ajouter à celui que nous avons déjà recommandé dans la section précédente : Cocoa Coding Guidelines d'Apple. En suivant les recommandations d'Apple et de Google, vous améliorerez la qualité de votre code.

Code et outils intéressants

F-Script

http://www.fscript.org/

F-Script, écrit par mon ami Philippe Mougin, est sans doute le premier logiciel que vous vous devez d'installer après Xcode. F-Script est un langage de script permettant d'introspecter les objets Cocoa et, grâce à un module spécial, il peut être injecté dans n'importe quelle application. Rien de tel pour apprendre le fonctionnement de Cocoa, déboguer vos applications, ou même permettre à l'utilisateur de l'utiliser comme langage de script pour automatiser l'application. F-Script est open-source, gratuit, et s'est placé second sur le podium des *Apple Design Award* à la WWDC 2006.

Cocoa Browser Air

http://numata.designed.jp/en/programming/cocoa-browser-air.html

Cocoa Browser est un petit logiciel de navigation de la documentation Cocoa et CocoaTouch. Il est très pratique car il vous donne un accès très simple et très rapide à l'ensemble de la documentation déjà copiée sur votre disque dur lors de l'installation de Xcode. Semblable à *AppKiDo*, il offre moins de fonctionnalités, mais est plus léger à l'utilisation.

AppKiDo

http://homepage.mac.com/aglee/downloads/appkido.html

AppKiDo est un logiciel de navigation de la documentation Cocoa. Semblable à *Cocoa Browser Air*, il offre plus de fonctionnalité mais est, en conséquence, légèrement plus lourd à l'utilisation.

OCMock

http://www.mulle-kybernetik.com/software/OCMock/

OCMock est une bibliothèque implémentant le modèle de conception appelé Mock. Un objet Mock permet de simuler n'importe quel comportement de manière totalement transparente pour le développeur. Par exemple, il est possible de demander à un objet Mock de se faire passer pour une instance de `NSString` et de lever une exception lorsqu'elle reçoit le message `length`.

Les objets Mock sont également très utiles pour tester des cas extrêmes difficilement réalisables en pratique (par exemple, insuffisance mémoire, disque dur plein, etc.).

Google Mac Developer Playground

http://code.google.com/mac/

De nombreuses applications et bibliothèques open-source Mac développées par Google sont disponibles depuis ce site.

Matt Gemmell

http://mattgemmell.com/source/

Matt Gemmell est un développeur indépendant ayant publié les sources de nombreuses de ses classes Cocoa, notamment des classes d'interface graphique assez intéressantes et utiles. Un grand nombre de ces classes sont utilisées par des freewares et sharewares.

Sparkle

http://sparkle.andymatuschak.org/

Sparkle est une bibliothèque gratuite et open-source semblable à la mise à jour de logiciel de Mac OS X. Vous pouvez ainsi proposer à vos utilisateurs de mettre votre logiciel automatiquement, et éviter les étapes successives d'une mise à jour manuelle.

SvnX

http://code.google.com/p/svnx/

Si vous êtes accroc à *Subversion* comme je le suis, vous ne vous contenterez sans doute pas de l'intégration de ce dernier dans Xcode, il vous faudra plus. La ligne de commande est déjà un bon point de départ, mais *SvnX* est un bon complément. C'est une interface graphique open-source et gratuite pour Subversion.

Versions

http://www.versionsapp.com/

Versions est un client Subversion pour Mac. Très évolué et fonctionnel, il est offre de nombreuses fonctionnalités pour un coût toutefois qui s'élève à 39 €.Versions a remporté un *Apple Design Award* à la WWDC 2009.

Isolator

http://willmore.eu/software/isolator/

S'il vous arrive d'avoir des périodes où la concentration est difficile et où la procrastination semble prendre la relève, Isolator vous sera sans doute d'une grande aide ! L'auteur, et par transitivité, son éditeur, regrettent de n'avoir pas découvert ce petit outil un peu tardivement.

Cappuccino

http://cappuccino.org/

Pour les experts Objective-C et Cocoa qui souhaitent développer des applications Web, il existe désormais une solution miracle, appelée Cappuccino : une quasi implémentation de Cocoa en JavaScript (d'où le nom Cappuccino : le café avec un peu de cacao). Les créateurs de Cappuccino ont même créé Objective-J : un langage très semblable à Objective-C, mais écrit en JavaScript.

GNUstep

http://www.gnustep.org/

GNUstep est un projet open-source tentant de réécrire toutes les bibliothèques de Cocoa (originellement, OpenStep) de sorte à rendre le code source écrit pour Cocoa multiplateformes. Bien que la tâche soit importante, le projet semble avancer lentement mais sûrement, et les applications compilées peuvent être exécutées sur Windows, Linux et de nombreux autres Unix.

Cocotron

http://www.cocotron.org/

Cocotron est un projet open-source tentant de fournir une implémentation multiplateformes de Cocoa. De même que GNUstep, le projet avance et semble suffisamment stable pour être utilisés en production par certains.

Index

A

B

C

D

E

F

G

I

O

INDEX

P

Q

R

S

T

U

V – W



LE GUIDE DE SURVIE

Objective-C 2.0

LE LANGAGE DE PROGRAMMATION IPHONE ET COCOA SUR MAC OS X

Ce *Guide de survie* est l'outil indispensable pour maîtriser Objective-C, le langage utilisé pour écrire les applications natives Mac OS X et iPhone. Vous y trouverez les bases d'Objective-C, ainsi que tout ce qu'il faut savoir pour bien gérer la mémoire, comprendre le système de notification et d'événements, migrer de la version 1.0 à 2.0, réaliser des tests unitaires et améliorer la qualité du code. Si vous venez d'autres langages, comme Java, C++, C# ou Python, il vous aidera à assimiler rapidement les spécificités d'Objective-C.

CONCIS ET MANIABLE

Facile à transporter, facile à utiliser — finis les livres encombrants !

PRATIQUE ET FONCTIONNEL

Une bibliothèque d'extraits de code et toutes les notions indispensables pour écrire un code Objective-C de qualité.

Pejvan Beigui est développeur et chef de projet dans une banque d'investissement américaine à Londres. Il a travaillé auparavant dans l'équipe des Relations développeurs d'Apple où il donnait des conférences et formait les développeurs et partenaires aux technologies Apple (Carbon, puis Cocoa).

Niveau : Intermédiaire / Avancé
Catégorie : Programmation
Configuration : Mac OS X

ISBN : 978-2-7440-4126-6



PEARSON
Pearson Education France
47 bis rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr